GETTING MORE FROM YOUR PET/CBM

Advanced Microcomputer Applications

E. A. Flinn • A. E. Hill • R. D. Tomlinson
(University of Salford)

Sigma Technical Press

Getting more from your PET/CBM

E A Flinn
A E Hill
R D Tomlinson

Sigma Technical Press

ISBN: 0 905104 23 4

Published by:
Sigma Technical Press, 5 Alton Road, Wilmslow, Cheshire, SK9 5DY.

Distributor: (UK and Europe):
John Wiley & Sons Ltd., Baffins Lane, Chichester, Sussex,
PO19 1UD.

Printed by:
UPS Blackburn Limited, 76-80 Northgate, Blackburn, Lancashire BB2 1AB.

Acknowledgements:
PET and CBM are trade marks of Commodore Business Machines Ltd.

Preface

This book had its origins in the programme of external short courses on microcomputers offered by the Department of Electronic and Electrical Engineering at the University of Salford. Several hundred people attend our courses each year, and the material presented here has been continuously honed and refined in the light of their comments and suggestions.

The contents of this book apply in general to all CBM microcomputers, although some sections are specific to 2000, 3000 and 4000 series PET's.

We suggest that, no matter what your own particular field of interest, you should read chapters 6 and 12, on programming style and interactive programming. If you are involved in business, administration, or other data handling applications you will probably find chapters 11 and 8 on disk data storage and the use of the CBM printer particularly relevant, while if you are a hobbyist, or involved in education, the sections on graphics and string handling may be of most interest.

We hope that you will find as much enjoyment and instruction in reading this book as we have found in writing it, and that you will continue to enjoy the fun and satisfaction of computing in the even more exciting years which lie ahead.

Finally, if the task of typing in these programs is just too overwhelming , a disk containing each of them is available. Please see the final page in this book.

E A Flinn
A E Hill
R D Tomlinson

Acknowledgements

Our sincere gratitude is due to Mrs. Nan Barker, without whose unfailingly cheerful assistance this book could not have been written; also (in alphabetical order) to Eric Brimble, Harold Foxcroft, Brian Lord, and Jay Smith, for their help and support in running the courses from which this book originated.

Contents

Example Programs

CHAPTER 1

Introduction to the PET/CBM Family

1 Introduction to the PET/CBM Family

The popular PET/CBM microcomputer was first introduced in 1978 and was one of the first desktop computers to incorporate a built-in monitor screen. Although the original concept remains largely unchanged there have been several significant changes to the operating system, the latest of which incorporates an upgraded BASIC called BASIC 4. BASIC 4 has given the PET faster string processing facilities and a set of disk operating commands which considerably simplify the use of the disk drive unit.

The earliest version of the PET had a small calculator-type keyboard and a built-in cassette unit. The very early machines operated with BASIC 1 and contained what are known as "Old ROMs". (A ROM is a Read Only Memory chip). These were quickly superseded by an improved BASIC operating system called BASIC 2. It was possible to upgrade the old BASIC 1 machines to BASIC 2 by installing a set of "new ROMs". BASIC 2 included a disk drive control system and a machine-code monitor, features which were not available in BASIC 1. The early machines were called the 2001 series (new ROMs were fitted to late models) and could be purchased with different amounts of user-accessible memory; 4K, 8K, 16K and 32K versions were produced. A large keyboard was introduced, initially for the 16K and 32K versions of the 2001 models; this feature has since become a standard fitting and the small keyboard models have been discontinued. Large keyboard machines require a separate cassette tape unit.

The 3000 series of PETs was introduced next, operating with BASIC 2, and the 4K version was discontinued. In late 1980 the 4000 series, operating with BASIC 4, was introduced. The first 4000 machines retained the 9" monitor screens and were essentially 3000 models equipped with BASIC 4 ROMs. Current 4000 series PETs have larger (12") monitor screens and incorporate some additional features, such as automatic repeat on the cursor control keys. All machines mentioned so far have a 40 column screen, which is a disadvantage for commercial users, particularly when word processing applications demand an 80 character per line printout.

The 8000 series, operating with BASIC 4, represents a recent extension to the PET/CBM range: an 80 column screen together with a 96K memory is now available and an improved disk drive (8050) system enables the machine to operate quite sophisticated business-orientated software packages. CBM have also in recent years introduced a low priced home computer range of machines called the VIC series which are designed to operate with a domestic TV receiver.

CHAPTER 2

Cursor Controlled Graphics and Animation

2 Cursor Controlled Graphics and Animation

The PET graphics facilities are built-in features of the keyboard and consist of 64 characters which are generated with the shift key, for the "standard" character set. This is a low resolution graphics facility but the wide range of available characters gives the PET quite a comprehensive sketching capability.

Close inspection of the screen reveals that each character is generated in an 8 × 8 matrix of points. Each point is called a pixel. If high resolution is required, additional hardware can be installed which allows the operator to program the control of any individual pixel or combination of pixels. (See Appendix C on extension ROM packages.)

Normally the screen can display a total of 1000 characters; 25 lines and 40 columns are available for text and graphics displays. Several different techniques are available for sketching and achieving animation effects. The cursor controls can be programmed into strings, while the POKE and PEEK commands can be used to place characters into specific screen positions and to identify the contents of any screen location.

2.1 SKETCHING WITH THE PRINT COMMAND

The simplest programmed sketching technique uses only the PRINT instruction to place strings of characters in specific screen locations. The following program is an example of this technique. The sketch can be produced directly and the program line number and PRINT statements can be inserted later, using the cursor controls and the editing facilities to avoid overwriting details of the sketch.

Example 2.1

```
100 PRINT"■"
110 PRINT
120 PRINT
130 PRINT
140 PRINT
150 PRINT
160 PRINT"          ▒▒▒
```

2.2 PROGRAMMED CURSOR CONTROL

The provision of cursor controls is a powerful and unique feature of Commodore BASIC. The cursor controls can be used in the direct mode and no doubt the reader is already familiar with this mode of operation. When programmed into a string, each control is represented by a special symbol which is printed in a program listing as shown in the following table.

Table 2.1

CURSOR CONTROL SYMBOLS

▪	CURSOR HOME
▪	CLEAR SCREEN
▪	CURSOR DOWN
▪	CURSOR UP
▪	CURSOR RIGHT
▪	CURSOR LEFT
▪	REVERSE CONTRAST ON
▪	REVERSE CONTRAST OFF

Cursor controls can be programmed into a string with the aid of the quotation marks key. This key acts like a switch, enabling the cursor movement to be programmed into a string. A second typed set of quotes, or the RETURN key, switches the PET back into direct cursor mode. The INSerT key also switches the PET into programmed cursor mode, thus allowing cursor controls to be edited into a string. After the inserted space or spaces have been filled the PET automatically returns to direct mode.

When the INST key has been used to insert extra spaces into a string the DELete key is also switched into programmed mode (indicated by a reverse contrast T), and the delete function cannot be operated until all inserted spaces have been filled. This means that it is not possible to erase directly an edited character if inserted spaces remain in the string, – a source of confusion for the beginner.

The following set of examples is intended to show how the programmed cursor controls can be used to produce graphics and text displays on screen.

Example 2.2

```
10 PRINT"▪"
20 PRINT"▪▪▪▪▪▪▪PET▪▪PET▪▪PET▪▪PET▪▪PET"
30 PRINT"▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪"
```

This program produces five "PETs" arranged in V formation with the third "PET" printed in reverse contrast. The cursor control character printed in line 10 clears the screen, and on line 20 seven "cursor right" controls are included before the first PET is printed. Line 30 merely shifts the cursor down screen so that the automatic READY does not mask the display.

The cursor automatically shifts to the right after printing a character on screen. This feature can be a source of confusion when generating graphics with the cursor controls, and the next program illustrates the point by printing out the "PETs" in a vertical line.

Example 2.3

```
10 PRINT"■"
20 PRINT"PET■■■PET■■■PET"
```

Three "left cursor" controls are necessary to allow for the automatic cursor advance.

The semicolon is used to concatenate strings and is a very important graphics control character. Strings of cursor controls can be concatenated in exactly the same way as strings of characters and the next program demonstrates this process.

Example 2.4

```
10 PRINT"■■■■■■■■■■■■■■■";
20 FOR I=1TO20:PRINT"▒";:NEXT I
30 FOR I=1TO6:PRINT"■▒";:NEXT I
40 FOR I=1TO20:PRINT"■▒";:NEXT I
50 FOR I=1TO6:PRINT"■▒";:NEXT I
60 PRINT"■■■■■■■NO PARKING";
70 PRINT"■■■■■■■■";
80 FOR I=1TO8:PRINT"■▒";:NEXTI
```

This program produces a signpost sketch similar to example 2.1. However, instead of using direct PRINT statements to produce the sketch, a series of concatenated strings are combined using FOR . . . NEXT loops. This is not meant to be the simplest way of producing the end result, which in this case would be the method used in 2.1 The program serves merely as an illustration of a technique which can be developed further to give increasing programmed control of a display or animation. The semicolons are included to concatenate the sequential printing of the character as the FOR . . NEXT loops are executed. Line 30 produces a vertical line, 6 characters long; the string contains a "cursor down" control followed by a "cursor left" control followed by the character. The controls move the cursor into a screen position immediately below the last printed character, so that a vertical line is created as the loop is executed. The automatic cursor advance only occurs when a character is printed and is not associated with the cursor positioning controls.

The SPC command may be used to replace some of the programmed cursor movements used in the last program. The argument of the SPC must be a positive number in the range 0 to 255 and the command moves the cursor through the specified number of spaces, to the right. The command is quite versatile because a variable or an expression containing variables may be used for the argument provided that numeric values in the allowed range are generated.

Lines 10 and 70 of the last example can thus be modified to:–

```
10 PRINT"■" SPC(207)
70 PRINT SPC(115)
```

SPC does not require quotation marks and can be used more than once in a program line.

For example

Example 2.5

```
10 PRINT"■"SPC(250)SPC(245)"MIDDLE"
```

The cursor move is started from the position directly before SPC is encountered; this command provides automatic concatenation.

2.3 ANIMATION USING PROGRAMMED CURSOR CONTROLS

Any character or group of characters can be printed in sequence with the aid of a FOR . . . NEXT loop to produce interesting animation effects. The cursor controls, together with the characters and an appropriate number of blanks (produced by the space key) are written as a string. Each pass through the loop causes the characters to be printed; the string is organised so that blanks are printed to erase the characters printed during the previous pass.

Example 2.6

```
100 PRINT"■"
110 FOR J=1 TO 39
120 PRINT" X■■";
130 FOR T=1 TO 50:NEXT T
140 NEXT J
```

This program moves a single character (the X symbol) from left to right on the top line of the screen through 39 successive print positions. The blank is positioned before the character so that after the first print the previous character can be erased. Each printed string contains two characters (the blank is a character) and so the last PRINT positions the X at the end of the line 40 spaces from HOME. The semicolon concatenates the sequential printing and the "cursor left" control positions the cursor over the character so that it can be erased by the blank during the following pass through the loop.

The maximum speed of the animation is determined by the string processing speed of the PET and if slower movement is required then a longer time delay loop can be incorporated.

The 4000 series of computers has a faster string processing facility than earlier machines. This improvement produces a speed increase of approximately 5 times. Hence the time delay loop becomes essential if you are working with a 4000 series PET.

The next program combines the previous program with a concatenated movement from right to left.

Example 2.7.

```
10 PRINT"■"
20 FOR J=1 TO 39
30 PRINT" X■■";
35 FOR T=1 TO 50:NEXT T
40 NEXT J
50 FOR J=1 TO 39
60 PRINT" ■■■X■■";
65 FOR T=1 TO 50:NEXT T
70 NEXT J
80 GOTO20
```

Line 60 in the previous listing produces the right to left animation. The string starts with the blank which erases the last character printed by the first movement. This is followed by two "cursor left" controls so that the X is printed in an adjacent position to the left of the blank. The last character in the string is a "cursor left"; this is included so that the cursor is positioned over the character, ready to erase it during the next pass. Vertical movement may be achieved using similar techniques:

Example 2.8

```
10 PRINT"■■■
```

```
210 PRINT"■"SPC(15)"■COURSE■"
220 FOR T=1 TO 100:NEXT T
230 GETA$:IFA$=" "THEN250
240 GOTO190
250 END
```

Lines 140 and 160 produce an animated figure (a man) which proceeds to "walk" across the screen from left to right. The string in line 140 generates the figure and is designed so that the blanks erase the trailing edge as left to right movement is achieved. In order to assist with the interpretation of this string a letter B has been inserted into the blank positions. Without this device it becomes difficult to distinguish some of the graphics characters which have been used to build up the figure. The method which has been used in this example to create the animation can be summarised as follows:–

1 The "man" is composed of four rows of characters.

2 The string (line 140) generates the top row first and then proceeds to build up rows 2, 3 and 4 in sequence.

3 Each row starts with a blank and after positioning the characters the cursor is moved to the start (left hand edge) of the next row.

4 After the figure is completed the cursor is moved back to the top row and positioned over the first character so that the next pass through the loop will erase the trailing edge. Line 160 is very similar to line 140; close inspection reveals that the only changes are associated with the graphics characters used to form the arms and legs. These are changed so that a "walking figure" can be simulated.

The GETA$ in line 230 provides an exit from the program by pressing the space key, and the next program can be added if more practice in animation techniques is required.

Example 2.11

```
250 PRINT"■■■■■■■■■";
260 FOR J=1 TO 18
270 PRINT" /■■■■■■■■■ "■    ■ ■■■■■■■■ ■PET■■■■■■■■  \ /■■■■";
280 PRINT" /■■■■■■■■■ "■    ■ ■■■■■■■■ ■PET■■■■■■■■ /   \■■■";
290 NEXT J
300 PRINT"■■■■"
310 PRINTSPC(15)"■\■ ■■■■■■■PET■"
320 FOR T=1 TO 20:NEXT T
330 PRINTSPC(15)"■ ■ ■■■■■■■PET■"
340 FOR T=1 TO 20:NEXT T
350 GOTO310
360 END
```

CHAPTER 3

PEEK and POKE Graphics

3 PEEK and POKE Graphics

CHARACTER CODES

The keyboard characters are recognised and stored in computer memory using two codes, the PET ASCII code and the PEEK/POKE code. Each of the codes may be used in BASIC programs with the aid of the following guide lines.

3.1 THE PET ASCII CODE

This code is a development of the ASCII (American Standard Code for Information Interchange) Code which is widely used in computer systems to specify characters and keyboard function operators. The code has been extended by CBM to include the characters and controls which are unique to the PET keyboard. Each character and control key is assigned a decimal number in the range 0 to 255, except for the reverse contrast characters which have no ASCII representation. Figure 3.1 shows the code numbers related to the large PET keyboard. PETs with small keyboards use the same ASCII character code but the keys are arranged in different locations.

The BASIC functions ASC and CHR$ use this code for character representation.

e.g. ?ASC("A") returns the number 65
and ?CHR$(65) returns the letter A.

CHR$ translates the decimal ASCII code number into the equivalent character and can be used to specify characters which cannot normally be used in strings.

e.g. ?CHR$(34); "PET"; CHR$(34)

returns the word "PET" in quotation marks on screen.

The ASCII code number for the quotation mark is 34; without the use of the CHR$ function it is not possible to program this symbol into screen text. It is also possible to program the cursor controls and the RETURN key with the aid of CHR$ and this function is used extensively when formatting text for the printer.

The ASC function finds useful graphics applications, some of which are illustrated later in this chapter.

3.2 THE PEEK/POKE CODE

This code is used in conjunction with the PEEK and POKE functions in order to POKE a character into screen memory or to PEEK a specific screen memory location in order to establish the contents. Each keyboard character is again assigned a decimal number in the range 0 to 255 but since in many cases the PET ASCII numbers are not the same as the PEEK/POKE code numbers, the use of the two codes requires careful management. Figure 3.2 relates the PEEK/POKE code to the large keyboard.

When using the PEEK and POKE functions to generate characters on screen it is necessary to know the screen locations in PET memory. The contents of the screen are stored in a block of 1000 memory locations, ranging from 32768 (top left) to 33767 (bottom right). These rather cumbersome numbers must be used as the argument in the PEEK/POKE functions.

e.g. POKE 33767,1

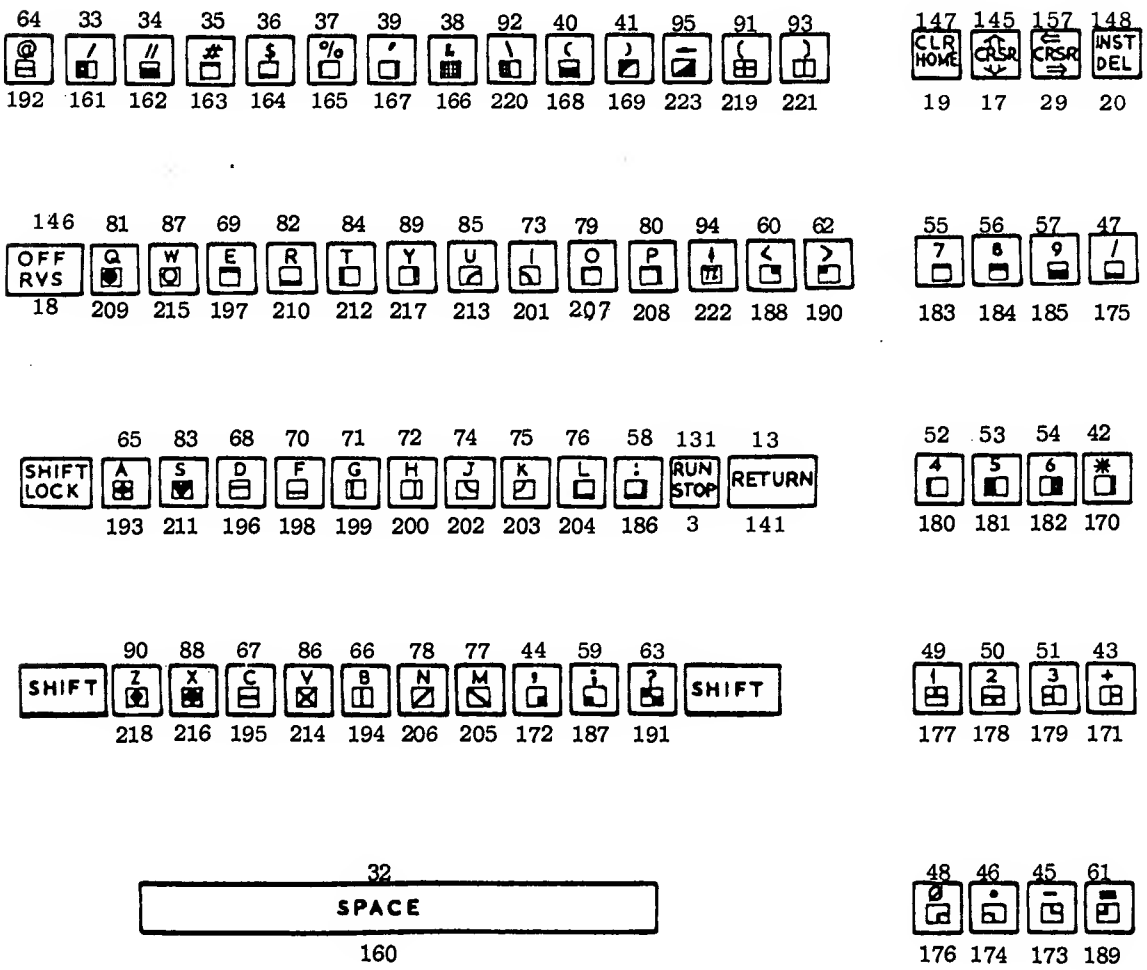
when used as a direct mode command will produce a letter A in the right hand bottom corner of the screen.

If this is followed by ?PEEK(33767) the number 1 is returned.

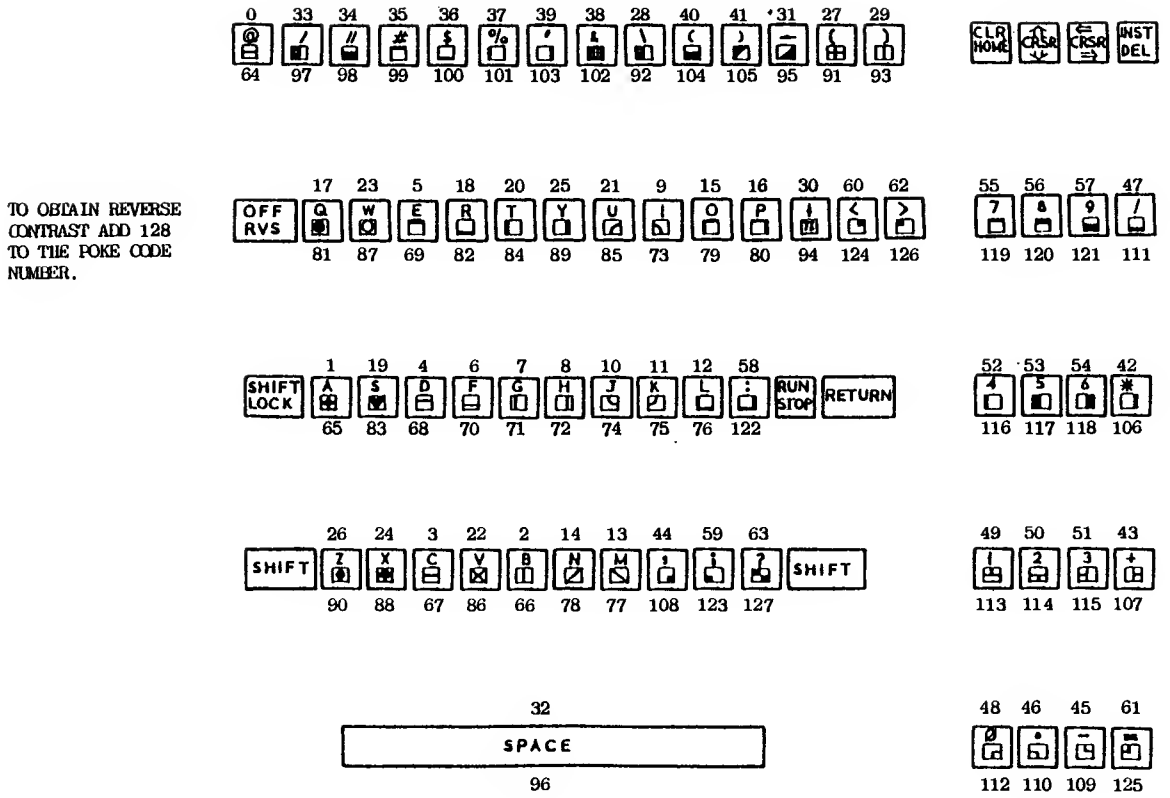
Any other screen location can be similarly addressed using other PEEK/POKE character code numbers to generate specific characters.

If the Alternate Character Set is used (POKE 59468,14) the codes now represent the new set of characters.

PET ASCII CODE NUMBERS



PEEK/POKE CODE NUMBERS



Code numbers reflect the new key interpretations

e.g. ?ASC("a") returns 65
and ?CHR$(65) returns "a"

Similarly, POKE33767, 1 now produces an a in the right hand corner of the screen and the command ? PEEK(33767) will return the number 1.

3.3 PEEK/POKE CODE CHARACTER RELATIONSHIPS

There are some useful relationships within the PEEK/POKE code which help the user to avoid constant reference to the tables.

The addition of 128 to any "normal contrast character" code number will produce the reverse contrast code number.

e.g. POKE33767, 1+128

returns a reverse contrast "A"

and if this is followed by ?PEEK33767 then 129 is returned.

Shifted characters are related in PEEK/POKE code to unshifted characters; the addition of 64 to the code number for any unshifted character produces the shifted version.

e.g. POKE33767, 22+64

returns a shifted V – the symbol X.

3.4 RELATIONSHIP BETWEEN THE PET ASCII AND THE PEEK/POKE CODES

It is possible to use the ASC function as the second argument in the POKE function and thus avoid reference to the code look-up table. Unfortunately, the relationship between the two codes is not the same for all keys so if the following procedure is used a little practice will be required.

The keyboard is divided into two blocks of keys;

Block 1 includes all the alphabet keys and several of the graphics keys, i.e. @ , \ , ← , [,] , and ↑ . All other characters form *Block 2*.

Block 1 relationships

The PEEK/POKE code number may be obtained by subtracting 64 from the ASCII code number.

e.g. POKE Z, ASC("N")– 64

generates the letter N at screen location Z. The code relationships for reverse contrast and shifted characters can also be incorporated into the POKE argument.

e.g. POKE Z,ASC("V")–64+64
or simply POKE Z,ASC("V")

returns the graphics symbol "X"

and POKE Z,ASC("N")–64+128

returns a reverse contrast N at screen position Z.

Block 2 relationships

For unshifted characters the ASCII and the PEEK/POKE code numbers are identical.

e.g. POKE Z,ASC("$")
generates the $ symbol and

POKE Z,ASC("5")
generates a 5 at screen position Z.

The rules for obtaining shifted and reverse contrast characters are unchanged, i.e.

POKE Z,ASC("5")+128

produces a reverse contrast 5.

3.5 GENERATING GRAPHICS WITH PEEK/POKE

The following programs in this chapter are examples of the use of the PEEK/POKE code functions for generating graphics displays.

The screen is organised into 40 columns and 25 rows, so that any location can be addressed using the formula

32768+40* (row number) + (column number)
0–24 0–39

The POKE command can be used with variables in both the arguments and the following exercise illustrates this facility, printing on screen a set of 38 characters (numbers 1 to 38 in the PEEK-POKE code). The shifted and reverse contrast characters are also generated in tabular form.

Example 3.1

```
100 REM   POKE CODE RELATIONSHIPS
110 PRINT"■": C=0
120 X=32808
130 FOR Y=1 TO 19
140 C=C+1
150 POKEX+Y+C,Y
160 POKEX+Y+C+80,Y+64
170 POKEX+Y+C+160,Y+64*2
180 POKEX+Y+C+240,Y+64*3
190 IFY=38THEN250
200 NEXT Y·
210 PRINT"1■■2■■3■■4■■■
220 X=32810+40*10
230 FOR Y=20 TO 38
240 GOTO 140
250 PRINT"■1■■■2■■■3■■■4■■"
260 PRINT"FIRST ROW-PET POKE CODE NUMBERS 1 TO38
270 PRINT"■SECOND ROW+64,THIRD +128,FOURTH +192.
280 GETA$:IFA$=""THEN 280
290 END
```

In line 150 the screen address is determined by the sum of the three parameters X, Y and C. A value of 32808 (2 lines down screen) has been chosen for X; this serves as a reference point for all subsequent screen positions. The FOR . . . NEXT loop serves to increment Y through the values 1 to 19 and C is also incremented so that two spaces are incorporated between the individual elements in the table. Y is also used to establish the PEEK/POKE code number for the second part of the POKE argument. Lines 150 to 180 generate four rows of characters. The second row, generated at line 160 is spaced two rows down screen from the first. This is accomplished by the addition of 80 to the POKE screen address; also the addition of 64 to the PEEK/POKE code number ensures that the set of shifted characters is produced. Sets of reverse contrast characters are generated at lines 170 and 180. The rows are labelled 1, 2, 3, 4 with the aid of the programmed cursor controls on lines 210 and 250. This process is repeated for Y values 20 to 38 in order to produce a second block of characters (with code numbers 20 to 38). The GET loop at line 280 serves only to suppress the READY at the end of the program but could be used to direct the program to a further set of characters if required.

The next program further illustrates the potential of the POKE command, generating a set of four rectangular borders on screen. The X variable is used sequentially to reduce the dimensions of the border and the GET at line 250 stops the execution of the program until an input from the keyboard is detected. Further features may be added to the end of this example. The GET holding loop provides the user with a convenient keyboard-controlled switch.

Example 3.2

```
100 PRINT"■":X=0
110 FOR K=1TO39-2*X
120 POKE32767+40*X+X+K,ASC("&")+64
130 NEXT K
140 FOR K=0TO23-2*X
150 POKE32807+40*X-X+40*K,ASC("&")+64
160 NEXT K
```

```
170 FOR K=1TO39-2*X
180 POKE33768-40*X-X-K,ASC("&")+64
190 NEXT K
200 FOR K=0TO23-2*X
210 POKE33728-40*X+X-40*K,ASC("&")+64
220 NEXT K
230 IFX=8  THEN250
240 X=X+2:GOTO110
250 GETA$:IFA$=""THEN250
```

The PEEK command can be used to scan the screen for character location information which can then be used to control the display. This function is demonstrated in the next program which reverses in sequence the contrast of all characters except the blank (POKE code number 32). The speed with which the PEEK and POKE functions operate within a BASIC program is limited and can only be increased by using machine code routines (see chapter 14).

Example 3.3

```
100 FOR X=32768 TO 33767
110 A=PEEK(X)
120 IF PEEK(X)>=128 THEN A=A-256
130 IF PEEK(X)=32 THEN 170
140 POKEX,A+128
150 GETA$:IF A$<>" "THEN 170
160 GETA$:IF A$<>" "THEN 160
170 NEXT
180 GOTO 100
```

Each screen location is PEEKed in turn and the PEEK/POKE code number assigned to the variable A at line 110. The reverse contrast character is POKEd back into the same location at line 140. Line 120 is required so that any character which is already in reverse contrast is generated with normal contrast. All reverse contrast characters have POKE code numbers $> = 128$ so if this is the case then A=A–256 produces normal contrast; code numbers greater than 255 are not allowed and without this device the program would crash at line 140 when the PEEK function identified a code number $>= 128$.

Lines 150 to 170 can be omitted but provide an interesting example of the use of the GET command. The space key is used to stop and start the execution of the program.

The POKE function can also be used to control (from the keyboard) the movement of a character or group of characters – a feature of many computer games.

Example 3.4

```
100 REM  CONTROLLED MOVE - GRAPHICS DEMO
110 REM
120 REM
130 REM  THE NINE DATA PAIRS SET UP THE
140 REM  DIRECTION OF THE MOVE.
150 REM
160 REM  2-DOWN, 8-UP, 6-RIGHT ETC
170 REM
180 REM  N IS THE INPUT AND R IS THE
```

```
190 REM  DIRECTION VARIABLE.
200 REM
210 FOR I=1 TO 9
220 READ N$(I),R(I)
230 DATA7,-41,8,-40,9,-39,6,1,3,41,2,40,1,39,4,-1,5,0
240 NEXT
250 L=33267
260 K=0
270 PRINT"■"
280 L=L+K
290 POKE L,65
300 GET X$
310 FOR I=1 TO 9
320 IFX$=N$(I) THEN 350
330 NEXT
340 GOTO 260
350 K=R(I)
360 GOTO 270
```

In the preceding program two arrays N(I) and R(I) are filled with data which links the movement of a specified character with keys 1 to 9 on the keyboard. The associated elements in the two arrays produce movement in the desired direction, e.g. N(2) contains 8 and R(2) contains –40, so that when an 8 is keyed in and detected by the GET at line 300, K is given the value –40. This value is then added to the screen address at line 280 and the character is regenerated one position up screen. Line 250 establishes an arbitrary start position and in line 290 the POKE code number 65 is an arbitrary character specification.

As a final example of the use of PET graphics facilities the framework of a simple shooting alley game is constructed. Most of the features which have been described in the previous two chapters are incorporated and it is left to the reader to produce his own variations.

Example 3.5

```
100 N=0
110 PRINT"■
120 FORI=1TO38
130 POKE32768+I,ASC("&")+64:NEXT
140 PRINT"■
150 FORI=1TO23
160 PRINT"■";:NEXT
170 FORI=1TO38
180 PRINT" ■■";
190 GETA$
200 IFA$="E"THEN250
210 FORJ=1TO20:NEXTJ
220 NEXT
230 PRINT" ";
240 GOTO140
250 PRINT"■";
260 N=N+1
270 FORI=1TO23
280 PRINT" ■■■";:NEXT
```

```
290 FORH=32769TO32807
300 T=PEEK(H)
310 IFT=102THEN390
320 NEXTH
330 PRINT"■
340 PRINT"YOU FIRED";N;"SHOTS"
350 PRINT"IF YOU WISH TO PLAY AGAIN TYPE Y"
360 GETA$:IFA$="Y"THEN100
370 IFA$=""THEN360
380 END
390 GOTO140
400 END
```

CHAPTER 4

Manipulating Text – String Handling

4 Manipulating Text – String Handling

INTRODUCTION

A string is composed of a sequence of alpha-numeric characters; that is, characters which need not be of a numerical nature. Thus, the PET's string facilities allow the programmer to use strings of alphabetic characters and graphical and other symbols in addition to pure numbers. This greatly increases the versatility of PET/CBM programs and is fundamental to programs which call for the manipulation of words and symbols.

A string variable is identified by the $ symbol and has many features in common with a numerical variable. Thus, typical string variables would be A$, BE$, C8$, etc. Strings may be assigned to string variables using the = sign, the string being enclosed in quotation marks.

Example 4.1

```
10 A$="DOG"
20 B$="CAT"
30 PRINT A$,B$
40 END
```

At an elementary level strings are very effective in personalizing the computer's response during a program.

Example 4.2

```
10 PRINT "WHAT IS YOUR NAME ?"
20 INPUT A$
30 PRINT "HELLO "A$".I AM PLEASED TO MEET YOU."
 |
 |
 |
100 PRINT "WELL DONE "A$".YOU ARE CORRECT."
 |
 |
 |
200 PRINT "I AM SORRY "A$" YOU ARE WRONG.TRY AGAIN."
```

4.1 COMPARISON OF TWO STRINGS

Two strings may be compared with each other by the use of the = sign. If such a comparison is incorporated in an IF . . . . THEN command this can form the basis of a simple question and answer quiz or teaching program.

Example 4.3

```
10 PRINT "■PLEASE TELL ME YOUR NAME"
20 PRINT
30 INPUT A$
40 PRINT "■HELLO "A$
50 FOR J=1 TO 1000 :NEXT J
60 B$="WHICH CITY IS THE CAPITAL OF FRANCE ?"
70 C$="PARIS"
80 PRINT "■"B$
90 PRINT
100 INPUT D$
110 PRINT
120 IF D$=C$ THEN PRINT "WELL DONE "A$:END
130 PRINT "■NO, IT IS NOT "D$". TRY AGAIN":GOTO 90
```

It should be noted that, in order for two strings to be judged to be the same, they must be truly identical. Thus, the characters must be in the same order and any spaces must also be in the corresponding places in each string. This can sometimes present a difficulty because a space at the start or the end of a string can be difficult to detect so that the PET may declare two strings to be different which, to the user, appear to be the same. A useful technique for detecting these invisible spaces is to evaluate the length of the string using the LEN command. Thus, the direct command PRINT LEN (A$) results in the number of characters contained in the string A$ being displayed. If this is apparently one character too many there is probably a hidden space present. Alternatively, the string can be displayed in reverse field in which case the hidden space becomes obvious.

Two strings may also be compared using the < and > symbols. However, this is misleading because it is not the string lengths which are being compared but the magnitudes of the ASCII values of the first characters of the string. If the first characters are the same then the next two are compared, and so on. Because the ASCII values for the alphabet characters ascend uniformly from A to Z, this feature can be very useful for the alphabetic sorting of lists of names and similar applications.

4.2 CONCATENATION

Two strings may be concatenated, i.e., joined together end to end, by the use of the + sign.

Example 4.4

```
10 A$="CAR"
20 B$="HIRE"
30 C$=A$+B$
40 D$=B$+A$
50 PRINT C$, D$
60 END
```

Note that there is no space between the two components of the concatenated string and that this must be introduced separately if required. Many strings may be joined in a single operation by this technique. It is often required that a string be broken down into smaller sub-strings, i.e. the reverse of the concatenation process. However, there is no such single process available and this must be achieved by the use of one of the following functions.

4.3 STRING HANDLING FUNCTIONS

The string handling functions LEFT$, RIGHT$ and MID$ are invaluable when manipulating strings. MID$ is particularly powerful.

LEFT$ (A$, N) will extract the first N characters from the string A$, starting from the left hand end of the string. These characters can then be assigned to a new string variable, if required.

Similarly, RIGHT$(A$, N) executes the same function but starting from the right hand end of the string.

Example 4.5

```
10 A$="ABCDEFGHIJ"
20 B$=LEFT$(A$,5)
30 C$=RIGHT$(A$,5)
40 PRINT B$,C$
50 END
```

Although extremely useful, the above commands are limited in that the starting points of the operation, namely the left and right hand ends of the string, cannot be changed. The only variable is the number of characters returned.

A more useful function is MID$. Thus, MID$(A$,N) extracts the Nth and all succeeding characters of the string A$ to give a result similar to that produced by RIGHT$. However, a second argument may be included in the expression, e.g. MID$(A$, N,P). This has a greater flexibility because both the starting point (N) in the string and the number of characters extracted (P) can be varied.

Example 4.6

```
10 A$="ABCDEFGHIJ"
20 B$=MID$(A$,5)
30 C$=MID$(A$,5,3)
40 PRINT B$,C$
50 END
```

4.4 SUBSCRIPTED STRING VARIABLES

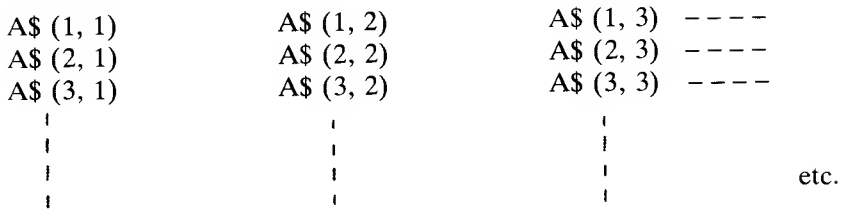
If many strings are to be used in a program it is a laborious and rigid process separately to assign each string to its string variable. This is made far easier and more flexible if subscripted string variables are used.

A single-dimensional subscripted string variable may take the forms A$(1), A$(23), etc. as compared with the non-subscripted forms A1$, A23$, etc.

A set of subscripted strings may be thought of as an array of variables. This array may be one-dimensional, as above, e.g.

A$ (1)
A$ (2)
A$ (3)
 |
 |

or it may be two-dimensional, e.g.



If the array does not exceed 11 subscripts in either direction (i.e. A$(0) . . . . A$(10) for a one-dimensional array) then the PET will immediately accept all the components of the array. If subscripts greater than 10 are required it is necessary to dimension the array before it is used. In effect, on receipt of a dimensioning (DIM) statement, the machine reserves sufficient space in its memory to accommodate the full array. Thus, a statement DIM A$(50) will allow subsequent use of subscripts up to 50. A two-dimensional array could be dimensioned as, say, DIM A$(12,15). Failure to dimension large arrays results in a break in the program and a BAD SUBSCRIPT ERROR message being displayed.

Subscripted strings are particularly powerful when used with FOR . . . NEXT loops.

Example 4.7

```
10 INPUT "ENTER NUMBER OF TERMS IN ARRAY";N
20 DIM A$(N)
30 B$="*"
40 FOR P=1 TO N
50 A$(P)=B$+A$(P-1)
60 NEXT P
70 FOR Q=1 TO N
80 PRINT A$(Q)
90 NEXT Q
100 END
```

This program fills a one-dimensional array A$(1) . . . . A$(N) with an increasing number of "*" characters. Note the use of the FOR . . . NEXT loop with the generalized variable A$(P) and also the use of concatenation in line 50 to increase automatically the number of "*" characters in each ascending variable. When the array has been filled completely, lines 70-90 cause the contents of the array to be displayed on the screen. The length of the delay between the start of the run and the start of the screen display gives a clear indication of the time taken in filling the array.

This program may also be used to demonstrate the alarming ease with which the memory can be filled when using these techniques. If N is set to 300, say, then an OUT OF MEMORY ERROR message is soon displayed. The point in the program at which this occurs will depend upon the memory size of the particular PET used. On a 32K machine it will be found that the last number of the array to be assigned is typically A$(245). If the direct command PRINT FRE(0) is entered to ascertain how much memory is still available it is found that there are typically only about 130 bytes unused, because of the large number and large size of the strings created.

To demonstrate the maximum permitted string length, line 30 may be altered to B$="**". When the program is re-run, with N=300, say, it now stops with a STRING TOO LONG ERROR message displayed. The last variable assigned is now A$(127) which contains 254 characters. The next variable, A$(128) would require 256 characters and would thus exceed the maximum permitted string character limit of 255.

It is clear that a two-dimensional array A$(N,P) will generally fill the memory at a lower value of the subscripts N and P than in the corresponding one-dimensional case A$(N) because the number of subscripted variables created is N multiplied by P. Thus, care must be exercised when using these arrays to ensure that the memory capacity is not exceeded. Higher orders of arrays may also be used, e.g. A$(N,P,Q) but they demand even greater care. For example, if N=P=Q=20 the completed three-dimensional array will have 8000 elements and, consequently, each element must on average not exceed four characters in length if the memory of a 32K machine is not to be over-filled.

4.5 LABELLING STRING CHARACTERS

One of the most interesting applications of string handling lies in the analysis and labelling of string characters. The most effective way to achieve this is to use the MID$ function together with subscripted string variables.

Example 4.8

```
10 A$="MANCHESTER"
20 N=LEN(A$)
30 FOR P=1 TO N
40 B$(P)=MID$(A$,P,1)
50 NEXT P
```

This program extracts each character in turn from the string A$ and assigns it to a position in the array B$(1) . . . B$(10). Thus B$(1)=M, B$(2)=A etc.

If required, these labelled characters may be re-assembled to reproduce the original string by concatenation.

Example 4.9

```
60 REM RE-ASSEMBLY
70 FOR Q=1TO N
80 B$=B$+B$(Q)
90 NEXT Q
100 PRINT B$
110 END
```

Note that no DIM statement is required in this program because here LEN(A$), and hence the highest subscript, is not greater than 10.

Once the characters of the string have been labelled they can be displayed individually in any order or, alternatively, they can be re-assigned to another array.

Example 4.10

```
10 INPUT "ENTER WORD";A$
20 N=LEN(A$)
30 DIM B$(N),C$(N)
40 FOR P=1 TO N
50 B$(P)=MID$(A$,P,1)
60 PRINT "B$("P")="B$(P)
70 NEXT P
```

```
80 FOR Q=1 TO N
90 C$(Q)=B$(N+1-Q)
100 PRINT "C$("Q")="C$(Q)
110 NEXT Q
120 FOR R=1 TO N
130 D$=C$(R)+D$
140 NEXT R
150 PRINT D$
160 END
```

Here lines 10-70 label the characters of any string A$ and display the labels individually. Lines 80–110 re-assign the B$(N) labels to a new array C$(N) in which the label numbers are reversed. Lines 120-150 re-assemble the original string from the new C$ array.

4.6 WORD LABELLING

The process of labelling the characters of a string may be extended, where appropriate, to the identification and labelling of the groups of string characters which make up the words in a sentence. To achieve this the above analysis program is modified to search for the spaces which separate adjacent words in the string. When such a space is identified the word is labelled as part of a subscripted string array.

Example 4.11

```
10 PRINT "□ENTER SENTENCE■■"
20 INPUT A$
30 N=LEN(A$):D=1:DIM B$(N):Z$=" "
40 C=1:REM C=NO.OF WORDS
50 FOR P=1 TO N
60 IF MID$(A$,P,1)=" "THEN 80
70 NEXT P
80 B$(C)=MID$(A$,D,P-D)
90 D=P+1
100 IF P<N THEN C=C+1:GOTO 70
```

Here, lines 50-70 examine each character of the string in turn, looking for a space. When a space is detected the program jumps to line 80 for the word to be labelled. The counter D is then set to P+1 in line 90 (P being the position of the character currently being examined). This has the effect that, when the next word is sent to line 80 for labelling, it is not accompanied by the previous word, nor by the space between them. Variable C is incremented after each word has been labelled to keep a record of the number of words in the string. When the complete string has been examined all the words of the string have been stored in the subscripted array B$(1) . . . . B$(C).

The words can later be re-assembled, as in lines 120-160.

Example 4.12

```
120 REM RE-ASSEMBLE
130 FOR Q=1 TO C
140 B$=B$+B$(Q)+Z$
150 NEXT Q
160 PRINT B$
```

Line 140 re-builds the original string from the labelled words which are concatenated together with spaces (Z$) inserted between them.

It is now possible to re-assemble the string with a different word order.

Example 4.13

```
180 REM RE-ASSEMBLE REVERSED
190 PRINT "■"
200 FOR R=1 TO C
210 C$=Z$+B$(R)+C$
220 NEXT R
230 PRINT C$
```

Here the original word order is reversed by the use of line 210 instead of line 140.

Another possible re-arrangement of the words is to sort them alternately into two sets, here denoted by printing alternate words in reverse field.

Example 4.14

```
250 REM SELECT ALTERNATE WORDS
260 PRINT "■"
270 J$="■":K$="■"
280 FOR S=1 TO C:IF INT(S/2)=S/2 THEN B$(S)=J$+B$(S)+K$
290 D$=D$+B$(S)+Z$
300 NEXT S
310 PRINT D$
320 END
```

In this case where the word label, i.e. the current subscript of B$(1) .... B$(C), is exactly divisible by two [so that in line 280, INT (S/2)=S/2], then the reverse field string, J$, is added to the word before it is displayed on the screen. Note that, although the reverse field is automatically switched off by the return at the end of line 280, it is good practice to include the "reverse field off" string K$ to avoid modifying any subsequent words unintentionally. This reverse field modification is not easily removed because there is no reverse of the concatenation procedure. If the word is needed again in its original form it is better to make a copy of it in some other string variable before J$ is added.

The complete program is shown below with the addition of lines 110, 170 and 240, which provide pauses until a key is pressed, to separate the different sections of the program.

Example 4.15

```
10 PRINT "■ENTER SENTENCE■■"
20 INPUT A$
30 N=LEN(A$):D=1:DIM B$(N):Z$=" "
40 C=1:REM C=NO.OF WORDS
50 FOR P=1 TO N
60 IF MID$(A$,P,1)=" "THEN 80
70 NEXT P
80 B$(C)=MID$(A$,D,P-D)
90 D=P+1
100 IF P<N THEN C=C+1:GOTO 70
110 PRINT "■":GET E$:IF E$=""THEN 110
120 REM RE-ASSEMBLE
```

```
130 FOR Q=1 TO C
140 B$=B$+B$(Q)+Z$
150 NEXT Q
160 PRINT B$
170 GET E$:IF E$=""THEN 170
180 REM RE-ASSEMBLE REVERSED
190 PRINT "■"
200 FOR R=1 TO C
210 C$=Z$+B$(R)+C$
220 NEXT R
230 PRINT C$
240 GET E$:IF E$=""THEN 240
250 REM SELECT ALTERNATE WORDS
260 PRINT "■"
270 J$="■":K$="■"
280 FOR S=1 TO C:IF INT(S/2)=S/2 THEN B$(S)=J$+B$(S)+K$
290 D$=D$+B$(S)+Z$
300 NEXT S
310 PRINT D$
320 END
```

An alternative ending to the program can be substituted after line 150:

Example 4.16

```
10 PRINT "■ENTER SENTENCE■■"
20 INPUT A$
30 N=LEN(A$):D=1:DIMB$(N),C$(N):Z$=" "
40 C=1:REM C=NO.OF WORDS
50 FOR P=1 TO N
60 IF MID$(A$,P,1)=" "THEN 80
70 NEXT P
80 B$(C)=MID$(A$,D,P-D)
90 D=P+1
100 IF P<N THEN C=C+1:GOTO 70
110 PRINT "■":GET E$:IF E$=""THEN 110
120 REM RE-ASSEMBLE
130 FOR Q=1 TO C
140 B$=B$+B$(Q)+Z$
150 NEXT Q
160 PRINT "■"
170 J$="■":K$="■":PRINT J$+B$+K$
180 PRINT "■■IDENTIFY THE PARTS OF SPEECH"
190 FOR R=1 TO C
200 PRINT "■WORD NUMBER "R"=";:INPUT F$
210 C$(R)=F$
220 NEXT R
230 PRINT "■"
240 FOR S=0 TO C-1
250 PRINT B$(S+1)SPC(15-LEN(B$(S+1)))"IS A "C$(S+1)
260 PRINT
270 NEXT S
280 END
```

This demonstrates how, once the words have been labelled, they may have further labels attached to them.

4.7 WORD SEARCHES

String functions may be used to identify specific words or characters in a sentence.

Example 4.17

```
10 DIM B$(80)
20 PRINT "□ENTER KEYWORD :- ";:INPUTX$
30 PRINT "□ENTER SENTENCE■■"
40 INPUT A$
50 N=LEN(A$):D=1
60 FOR P=1 TO N
70 IF MID$(A$,P,1)=" " THEN 90
80 NEXT P
90 IF MID$(A$,D,P-D)=X$ THEN 150
100 D=P+1
110 IF P<N THEN 80
120 PRINT "□■THE KEYWORD IS NOT PRESENT■"
130 PRINT "■■IN THIS SENTENCE■"
140 GOTO 160
150 PRINT "□■THIS SENTENCE CONTAINS THE KEY WORD■"
160 FOR J=1 TO 2000:NEXT J
170 GOTO 30
```

Here, the same word-analysis procedure as used previously is employed in lines 50-100 with the difference that, instead of a word being labelled and placed in a subscripted array, it is simply compared with the keyword X$ in line 90. A simple modification could be added to this program to count the number of times the keyword occurs.

An alternative word-identification technique allows a series of data statements to be scanned for a keyword. This may be extended to the association of the identified keyword with preceding or subsequent data.

Example 4.18

```
10 INPUT "□ENTER KEY CONDITION";X$
20 READ A$:IF A$="*" THEN 60
30 IF LEFT$(A$,1)="-" THEN Z$=A$:Y=Y+1: REM Y=TOTAL PATIENT COUNT
40 IF A$=X$ THEN 80
50 GOTO 20
60 IF D=0 THEN PRINT "□■KEY CONDITION NOT PRESENT■■■":GOTO 140
70 GOTO 110
80 IF D=0 THEN PRINT"□■"X$" PATIENTS■"
90 PRINT "■PATIENT:"Z$:D=1:X=X+1:REM X=COUNT OF KEY CONDITION PATIENTS
100 GOTO 20
110 PRINT "■■THAT IS ALL■"
120 PRINT "■NUMBER OF "X$" PATIENTS="X
130 PRINT "■NUMBER OF PATIENTS ON FILE="Y
140 PRINT "■■■PRESS THE SPACER TO CONTINUE■"
150 GET E$:IF E$<>" " THEN 150
160 RESTORE:D=0:X=0:Y=0:Z$="":GOTO 10
170 DATA -COCKROFT S.M.,EPILEPSY,-COLEMAN S.,RHEUMATISM,BRONCHITIS
180 DATA DIABETES,-COUGHLAN F.W.,ANGINA,STROKE,-CROMPTON A.,RHEUMATISM
190 DATA -DAWSON C.,LUMBAGO,-DENNIS H.,STROKE,-DOLAN J.C.,CANCER,THROMBOSIS
200 DATA -DRIVER J.,BRONCHITIS,DIABETES,-DURHAM E.,CANCER,-ECCLES C.A.,ANGINA
210 DATA -EMERY G.,MIGRAINE,LARYNGITIS,-FELL P.H.,ANGINA,-FIELDING E.,LUMBAGO
220 DATA -FITCH L.,PNEUMONIA,BRONCHITIS,-GATCHET A.,BRONCHITIS,THROMBOSIS
230 DATA -GARTSIDE F.,STROKE,-GEE B.,CANCER,-GILES G.S.,MIGRAINE,RHEUMATISM
```

```
240 DATA -GROAN J.P.,EPILEPSY,RHEUMATISM,DIABETES,ANGINA,LUMBAGO,STROKE,CANCER
250 DATA BRONCHITIS,LARYNGITIS,MIGRAINE,PNEUMONIA,THROMBOSIS,PLEURISY,SCIATICA
260 DATA -HAMPSON I.,BRONCHITIS,-HARLICK G.,DIABETES,THROMBOSIS,-HARRIS V.E.
270 DATA PLEURISY,-HIBBERT H.,RHEUMATISM,SCIATICA,-HOLLINDRAKE J.,STROKE
280 DATA -JACKSON N.,CANCER,LUMBAGO,-JONES R.,CANCER,-KELLY G.,DIABETES,STROKE
290 DATA -KNOWLES H.,BRONCHITIS,-LAINE A,LUMBAGO,PNEUMONIA,BRONCHITIS,ANGINA
300 DATA -LAWTON T.,SCIATICA,-LONGDEN E.,THROMBOSIS,ANGINA,-MACKRELL H.,STROKE
310 DATA -PARKER J.,CANCER,-PEARSON A.,DIABETES,-QUINN P.J.,LARYNGITIS,ANGINA
320 DATA -RAMSDEN S.,BRONCHITIS,PNEUMONIA,-WILLIS J.T.R.,DIABETES,RHEUMATISM
330 DATA *
```

This is a program to identify the patients in a doctor's files who suffer from a particular illness. The patients' names are identified by a preceding hyphen. The keyword is supplied in line 10 and stored as X$. The data statements containing the patients' names and their various ailments are scanned in lines 20-50. Each time a data statement preceded by a hyphen is encountered the patient count, Y, is incremented and the patient's name is stored temporarily in Z$ (line 30). When the key illness X$ is detected the current value of Z$ gives the name of a patient suffering from this illness. This name is displayed on the screen and the search continues until the data entry "*" is read which terminates the search. D is a flag which is set the first time the keyword is identified and is used to dictate the form of the screen display. Note the necessity for the command RESTORE in line 160 before the data base can be re-entered for a new search. Note also that it is not necessary to use subscripted strings with their associated complication of DIM statements and their voracious use of memory space. This is because the data are not labelled and stored for future use, the temporary store Z$ being all that is required.

In this simple form of search program the process is limited only by the size of the available memory. Typically 1000 patients and their ailments can be handled on a 32K machine. A more realistic approach would be to store the data base on a disk file, sections of which would be periodically transferred to the PET for the searching process.

As a final exercise it is interesting to use some of the techniques already discussed to write an "anagrammatizer" program in which a word is picked at random from a data base and scrambled. One possible solution is shown below together with additional program statements which make it the basis of a game whereby the player's guess is compared with the original, unscrambled, word.

Example 4.19

```
10 PRINT "□              ANAGRAM TIME !"
20 DIM W$(185):DIM R(30):DIM B$(30):DIM C$(30)
30 REM ASSIGN DATA WORDS TO W$() ARRAY
40 FOR P=1 TO 185:REM  185 = NUMBER OF WORDS IN THE DATA BASE
50 READ W$(P)
60 NEXT P
70 PRINT "□"
80 Z=RND(-TI):    REM RANDOM SEED
90 REM CHOOSE A WORD TO ANAGRAMMATIZE
100 Z=INT(186*RND(1))
110 A$=W$(Z)
120 REM LABEL LETTERS OF WORD AS B$(1..N)
130 N=LEN(A$)
140 FOR Q=1 TO N
150 B$(Q)=MID$(A$,Q,1)
160 NEXT Q
170 C$=""
180 PRINT"□QUIET PLEASE !■■■■■■■■■■■■■■■■■■I AM ANAGRAMMATIZING
190 REM CHOOSE A RANDOM LETTER POSITION
200 FOR S=1 TO N
210 R=INT((N+1)*RND(1))
220 IF R=0 THEN 210
230 REM SET FLAG W IF THIS POSITION HAS BEEN USED BEFORE
```

```
240 FOR T=1 TO N
250 IF R=R(T)THEN W=1
260 NEXT T
270 REM POSITION ALREADY USED.TRY AGAIN
280 IF W=1 THEN W=0:GOTO 210
290 REM ASSIGN STH. POSITION IN SCRAMBLED WORD TO RTH. LETTER IN
300 REM ORIGINAL WORD
310 C$(S)=B$(R)
320 REM RECORD THAT RTH. POSITION HAS  NOW BEEN USED
330 R(S)=R
340 NEXT S
350 REM RE-ASSEMBLE SCRAMBLED WORD
360 FOR U=1 TO N
370 C$=C$+C$(U)
380 R(U)=0
390 NEXT U
400 REM TRY AGAIN IF SCRAMBLED WORD=ORIGINAL
410 IF C$=A$ THEN 170
420 REM PRINT SCRAMBLED WORD
430 PRINT"■THE ANAGRAM IS ■"C$"■"
440 PRINT"■■■WHAT IS THE ORIGINAL WORD ?"
450 PRINT"■■":INPUT D$
460 IF D$=A$ THEN PRINT "■CORRECT!WELL DONE":GOTO 630
470 PRINT "■NO!THAT IS NOT CORRECT"
480 PRINT "■■■■WOULD YOU LIKE TO TRY IT AGAIN ?"
490 PRINT "■(PLEASE ANSWER Y OR N)
500 GET E$:IF E$="" THEN 500
510 IF E$="Y" THEN 540
520 IF E$="N" THEN 590
530 GOTO 500
540 PRINT "■■■SHALL I RE-ARRANGE THE WORD ? (Y OR N)"
550 GET F$:IF F$="" THEN 550
560 IF F$="Y" THEN PRINT "■":GOTO 130
570 IF F$="N" THEN PRINT "■":GOTO 410
580 GOTO 550
590 J$="■"
600 F$=J$+A$
610 PRINT "■ALL RIGHT.I WILL TELL YOU"
620 PRINT "■THE WORD WAS "F$
630 PRINT "■■■■WOULD YOU LIKE ANOTHER WORD ?"
640 PRINT "■(PLEASE ANSWER Y OR N)"
650 GET E$:IF E$="" THEN 650
660 IF E$="Y" THEN 100
670 IF E$="N" THEN PRINT "■THANK YOU FOR PLAYING■■■■■■■■■■":END
680 GOTO 650
690 DATA ABSENCE,ACCEPT,ACCIDENTALLY
 |
 |
 |
 |
```

and so on. DATA statements containing an additional 182 words are added to make a total of 185 words (line 40).

CHAPTER 5

Sorting Words and Numbers

5 Sorting Words and Numbers

INTRODUCTION

Sorting lists of numbers into numerical order or lists of names into alphabetical order is a dreary task which is better performed by a computer. There are many different sorting routines available in which, generally, complexity may be traded off against speed.

5.1 BUBBLE SORT

The simplest and probably the most widely used routine is the bubble sort. This is so called because the smallest number appears to 'bubble' to the top of the list. Bubble sorts are very widely employed because of their simplicity.

In a numerical bubble sort a list of numbers is repeatedly scanned from top to bottom. On each pass each number is compared with the one below it. If it is smaller than the one below, it is left alone; if it is larger the two numbers are interchanged and the scan proceeds to the next number, (which, of course, will be one of the numbers just interchanged.). It can be seen that, after the first pass, the largest number will be at the bottom of the list. A second pass will leave the next largest number in the next to the bottom position and so on. At first it appears that to be certain that a list of N numbers has been completely sorted N–1 passes would be needed. However, this lengthy procedure may be shortened. It is not necessary to include every number in each pass. Thus, it would be pointless to look at the last number on the first pass because it is already in place. Similarly for the next to the last number on the second pass etc. Thus, each pass contains one less comparison than the previous one.

Also, if the original list is randomly arranged, then by chance all the numbers may be in order before the (N–1)th pass, in which case further passes are unnecessary. Thus, if on any pass it is found that no interchanges were necessary, the sort may be terminated.

A simple bubble sort is shown in Example 5.1, where ten numbers are sorted into numerical order. First the numbers are read from data statements into the array A(1) . . . . . A(10) and at the same time they are printed on the screen (line 40). The sort proper starts in line 60 where X is set to N–1, the number of passes expected. In line 80 Y is set up representing the number of comparisons made in each pass and on each successive pass the range of Y is decremented by 1. The number of comparisons made is counted and stored in the count variable C in line 90. Adjacent numbers are compared in line 100. If they are in order then a jump is made to 160 to start the next comparison. If not, an interchange is made in 130-150, a flag Z is set to 1 to show that an interchange has occurred and the interchange counter E is incremented. At the end of the pass line 170 checks flag Z. If it is zero then the data must be in order and the sort can be terminated. If it is a 1 then the next pass is initiated.

If not previously terminated the sort is ended when the (N-1)th pass has been completed. Although not strictly part of the sort, the timing variables T1 and T2 and the comparison and interchange counters C and E give useful indications of the efficiency of the process.

Example 5.1

```
1 REM *** BUBBLE SORT ***
10 READ N :REM NUMBER OF ENTRIES
20 DIMA(N)
30 PRINT "■■UNSORTED LIST:-■"
40 FOR X=1 TO N:READ A(X):PRINTA(X);:NEXT X
50 T1=TI
60 FOR X=1 TO N-1
70 Z=0
80 FOR Y=1 TO N-X
90 C=C+1
100 IF A(Y+1) >= A(Y) THEN 160
110 Z=1 :REM FLAG SET IF DATA NEEDS TO BE INTERCHANGED ON THIS PASS
120 E=E+1
130 W=A(Y)
140 A(Y)=A(Y+1)
150 A(Y+1)=W
160 NEXT Y
170 IF Z=0 THEN 200:REM DATA IN ORDER?
180 NEXT X:REM START ANOTHER PASS
190 REM SORT COMPLETE
200 T2=TI
210 PRINT:PRINT"■SORTED LIST:-■"
220 FOR X= 1 TO N:PRINT A(X);:NEXT X
230 PRINT
240 PRINT"■■NUMBER OF COMPARISONS  =";C
250 PRINT"■NUMBER OF INTERCHANGES =";E
260 T=.01*INT(100*(T2-T1)/60)
270 PRINT"■TIME TAKEN ="T"SECONDS"
280 DATA 10
290 DATA 8,6,6,3,7,2,5,4,9,0
300 END
```

As can be seen, the bubble sort is very quick for relatively small lists, but as the size of the list to be sorted increases, the sorting time becomes much longer. This is demonstrated in Example 5.2 which is exactly the same sort as the previous example, but this time sorting an array of randomly generated numbers. If N is set to 100, a typical analysis at the end of the sort might be:–

NUMBER OF COMPARISONS	=	4935
NUMBER OF INTERCHANGES	=	2475
TIME TAKEN	=	170.53 SECONDS

Because of the ability of the bubble sort to cut short its routine if the list becomes ordered before the (N-1)th pass, it is difficult to estimate the time required for the sort. Bubble sorts perform best when the data are already partially ordered. Thus, if the data are ordered after the first pass, only (N-1) comparisons have to be made. But, if the list is in reverse order, the (N-1) comparisons will only place the highest value at the end of the list and further passes of length (N-2), (N-3), ... 2, 1 are needed. So the maximum sort effort is N(N-1)/2. On average, for a randomly ordered list, the sort effort will be N(N-1)/4, or, for large N, approximately $N^2/4$.

Example 5.2

```
1 REM *** BUBBLE SORT ***
10 PRINT"■                ■BUBBLE■ ■SORT■"
20 INPUT "■■HOW MANY NUMBERS TO SORT ";N
30 DIM A(N):E=0:C=0
40 :
50 REM*** PUT UNSORTED NUMBERS IN ARRAY ***
60 :
70 GOSUB 550
80 :
90 REM *** PRINT THE UNSORTED ARRAY ***
100 :
```

```
110 PRINT "■THE UNSORTED ARRAY IS :■"
120 :
130 GOSUB 600
140 :
150 REM *** SORT IT ***
160 :
170 GOSUB 340
180 :
190 PRINT "THE SORTED ARRAY IS :■"
200 :
210 GOSUB 600
220 :
230 PRINT"              ■BUBBLE■ ■SORT■■"
240 PRINT"NUMBER OF COMPARISONS  =";
250 B3=C:GOSUB 740
260 PRINT"NUMBER OF INTERCHANGES =";
270 B3=E:GOSUB 740
280 T=.01*INT(100*(T2-T1)/60)
290 PRINT"■TIME TAKEN ="T"SECONDS"
300 END
310 :
320 REM *** BUBBLE SORT ***
330 :
340 T1=TI
350 FOR X=1 TO N-1
360 Z=0
370 FOR Y=1 TO N-X
380 C=C+1
390 IF A(Y+1) >= A(Y) THEN 450
400 Z=1 :REM FLAG SET IF DATA NEEDS TO BE INTERCHANGED ON THIS PASS
410 E=E+1
420 W=A(Y)
430 A(Y)=A(Y+1)
440 A(Y+1)=W
450 NEXT Y
460 IF Z=0 THEN 490:REM DATA IN ORDER?
470 NEXT X:REM START ANOTHER PASS
480 REM SORT COMPLETE
490 T2=TI
500 RETURN
510 :
520 :
530 REM*** FILL ARRAY WITH RANDOM NUMBERS ***
540 :
550 FOR K=1 TO N:A(K)=INT(100*RND(0)):NEXT K
560 RETURN
570 :
580 REM *** PRINT ROUTINE FOR TEN COLUMNS ***
590 :
600 FOR K=1 TO N
610 Q=K-10*INT((K-1)/10)
620 PRINT TAB(4*(Q-1));
630 IF A(K)<10 THEN PRINT" ";
640 PRINT A(K);
650 IF Q>=10 THEN PRINT
660 NEXT K
670 PRINT"■"
680 RETURN
690 :
700 :
710 REM *** COLUMN ALIGNMENT ROUTINE ***
720 :
730 :
740 B1=0:B2=B3
750 IF B2<1 THEN 770
760 B1=B1+1:B2=B2/10:GOTO 750
770 PRINT SPC(6-B1);B3
780 PRINT
790 RETURN
```

This example has been written in the form of a main program with numerous subroutines because it is in this form that sort routines are most likely to be used and encountered. Other subroutines, apart from the sort subroutine, are concerned with the generation, formatting and presentation of the data.

Numerical sorting techniques may easily be extended to the sorting of words into alphabetic order. Two word strings may be compared using the operators > and <. These operators compare the magnitudes of the ASCII values of the first characters of the strings. If these are the same then the next two are compared and so on. Because the ASCII values of the alphabet characters ascend uniformly from A to Z this gives a true alphabetic sort. Thus CAN will be judged to be "less than" (<) CAR which in turn will be "less than" (<) CAT. Care must be taken when including punctuation marks and spaces in the strings because these will also be included in the comparison. However, as these all have ASCII values less than A the sort is not usually upset.

Example 5.3 is essentially the same bubble sort as in the previous two examples but modified to deal with strings. Thus, it is strings which are read from the data statements in 20-40, strings which are compared in 90 and strings which are interchanged in 110-130. The time taken for the sort is virtually the same as the time taken to sort the same number of numerical elements.

Example 5.3

```
1 REM *** ALPHA BUBBLE SORT ***
10 READ N :REM NUMBER OF ENTRIES
20 DIM A$(N)
30 PRINT "□UNSORTED LIST:-■"
40 FOR X=1 TO N:READ A$(X):PRINTA$(X):NEXT X
50 T1=TI
60 FOR X=1 TO N-1
70 Z=0
80 FOR Y=1 TO N-X
90 IF A$(Y+1) >= A$(Y) THEN 140
100 Z=1 :REM FLAG SET IF DATA NEEDS TO BE INTERCHANGED ON THIS PASS
110 W$=A$(Y)
120 A$(Y)=A$(Y+1)
130 A$(Y+1)=W$
140 NEXT Y
150 IF Z=0 THEN 170:REM DATA IN ORDER?
160 NEXT X:REM START ANOTHER PASS
170 REM SORT COMPLETE
180 T2=TI
190 PRINT:PRINT"■" SPC(20)"SORTED LIST:-■"
200 FOR X= 1 TO N:PRINT SPC(20) A$(X):NEXT X
210 T=.01*INT(100*(T2-T1)/60)
220 PRINT"■TIME TAKEN ="T"SECONDS"
230 DATA 18
240 DATA EMERY G.,DOLAN J.C.,HARRIS V.E.,RAMSDEN S.,FIELDING E.,MACKRELL H.R.
250 DATA PEARSON A.,HARRIS V.J.,LONGDEN E.,HARLICK G.,LAWTON J.,PARKER J.
260 DATA COLMAN S.,DENNIS H.,FELL P.H.,FITCH L.,QUINN P.,HOLLINDRAKE J.
```

Example 5.4 is again a bubble sort but this time the data are extended to cover three subjects or fields instead of one. Thus, the first field is a name, the second an age and the third some sort of mark or grade signified by the letters A to D. The data are read into a two-dimensional array A$(N, F) in lines 30-70, where N is the number of multi-field elements to be sorted and F is the number of fields. Thus, in this case, A$(1,1) is SMITH, A$(1,2) is 24 and A$(1,3) is B etc. The contents of this array are displayed in 110-160. The bubble sort is then entered at 180 and in 210 the elements are compared. However, the comparison can be made in either the 1st, 2nd or 3rd field, depending upon the value of K entered in line 90. If, as a result of the comparison, an interchange is required then it is essential that all the fields of the relevant element are interchanged together. This is achieved by a FOR . . . NEXT loop in conjunction with the normal interchange instructions in 220-260. When the sort has been completed the sorted array is displayed. To repeat the sort with a different key field K following the

GET hold in 430, the data statements are read again, this being the easiest way to restore the original A$( ) array. This necessitates a RESTORE in 440. (None of this would be needed if the unsorted array were not displayed.)

Example 5.4

```
1 REM *** 3 FIELD BUBBLE SORT ***
10 READ N,F:IF Z2=1 THEN 30
20 DIM A$(N,F)
30 FOR X=1 TO N
40 FOR J=1 TO F
50 READ A$(X,J)
60 NEXT J
70 NEXT X
80 PRINT"□ENTER KEY FIELD (1,2,3 ETC.)
90 GET K:K=INT(K):IF K=0 OR K>F THEN 90
100 :
110 PRINT "□UNSORTED LIST"
120 PRINT "──────────■"
130 FOR X=1 TO N
140 FOR J=1 TO F
150 PRINT A$(X,J)"      ";:NEXT J
160 PRINT:PRINT:NEXT X
170 T1=TI
180 FOR X=1 TO N-1
190 Z1=0
200 FOR Y=1 TO N-X
210 IF A$(Y+1,K) >= A$(Y,K) THEN 280
220 FOR P=1 TO F
230 W$=A$(Y,P)
240 A$(Y,P)=A$(Y+1,P)
250 A$(Y+1,P)=W$
260 NEXT P
270 Z1=1
280 NEXT Y
290 IF Z1=0 THEN 310
300 NEXT X
310 T2=TI
320 PRINT
330 PRINT "SORTED LIST"
340 PRINT "──────────■"
350 FOR X=1 TO N
360 FOR J=1 TO F
370 PRINTA$(X,J)"      ";:NEXT J
380 PRINT:PRINT:NEXT X
390 T=.01*INT(100*(T2-T1)/60)
400 PRINT "■"SPC(250)SPC(250)SPC(122)"■KEY FIELD="K
410 PRINT SPC(102)"■T(SECONDS)="T
420 PRINT "■■■■■PRESS ANY KEY TO CONTINUE■"
430 GET G$:IF G$="" THEN430
440 RESTORE:Z2=1:GOTO 10
450 DATA 4,3
460 DATA SMITH,24,B,JONES,43,A,ADAMS,65,D,BROWN,42,C
```

5.2 INDEXED SORT

An alternative technique for sorting multi-field arrays is to assemble a control array in parallel with the data array. Numbers in the control array are manipulated as a result of sorting comparisons made in the key field of the data array. At the end of the sort the control array determines the order in which the elements of the data array are presented. This technique avoids the necessity to move all the fields in the data array together during interchanges. Example 5.5 demonstrates this approach using the same data as in the previous exercise. The data are read into the array A$( ) as before but in lines 110 to 130 the key field, K, is stripped off into another array C$( ). The sub-routine at 330 assembles the control array Y( ). The contents of this array indicate the positions in the key field array C$( )of the smallest element, the next smallest element and so on. Thus, if K is 2, and hence C$(1)=24, C$(2)=43, C$(3)=42 and C$(4)=65 then, on leaving the subroutine, Y(1)=1, Y(2)=3, Y(3)=2 and Y(4)=4. This array now controls the order of print-out of the main array A$( ) in 180-220.

Example 5.5

```
1 REM *** 3 FIELD INDEXED SORT ***
10 READ N,F:DIM A$(N,F),C$(N),Y(N)
20 FOR X=1 TO N
30 FOR J=1 TO F
40 READ A$(X,J)
50 NEXT J
60 NEXT X
70 PRINT"□ENTER KEY FIELD (1,2,3 ETC.)
80 GET K:K=INT(K):IF K=0 OR K>F THEN 80
90 :
100 T1=TI
110 FOR X=1 TO N
120 C$(X)=A$(X,K)
130 NEXT X
140 GOSUB 330
150 T2=TI
160 PRINT "□SORTED LIST"
170 PRINT "──────────■"
180 FOR X=1 TO N
190 FOR J=1 TO F
200 PRINT A$(Y(X),J);"      ";:NEXT J
210 PRINT:PRINT
220 NEXT X
230 PRINT"■■■KEY FIELD ="K
240 T=.01*INT(100*(T2-T1)/60)
250 PRINT"■■T(SECONDS) ="T
260 PRINT"■■■■■PRESS ANY KEY TO CONTINUE■"
270 GET G$:IF G$ ="" THEN 270
280 :
290 GOTO 70
300 :
310 REM *** ASSEMBLE CONTROL ARRAY ***
320 :
330 FOR P=1 TO N
340 M=1
350 FOR Q=1 TO N
```

```
360 IF C$(P)>C$(Q) THEN M=M+1
370 IF C$(P)=C$(Q) AND P>Q THEN M=M+1
380 NEXT Q
390 Y(M)=P
400 NEXT P
410 RETURN
420 :
430 DATA 4,3
440 DATA SMITH,24,B,JONES,43,A,ADAMS,42,D,BROWN,65,C
```

N.B. In both of the two previous examples the arrays to be sorted are string arrays even though some of the fields are numerical. Thus, if a numerical field is the key field, it will be sorted as a string field. This is satisfactory as long as all the elements have the same number of digits. If this is not so then sorting errors will occur. Thus 8 would be judged to be greater than 42 because, in a string comparison, only the first characters are compared initially. In such cases it is therefore necessary to include leading zeros e.g. 08 so that all the elements are of the same length.

5.3 INSERTION SORT

As the name implies, the insertion sort works on the principle of scanning a list of elements and inserting a new element in its correct position. Thus, elements from one array may be inserted into another array. Alternatively, elements may be fed in from the keyboard as input statements and inserted into their correct position, often in a time less than that required to type in an entry. A simple insertion sort is shown in Example 5.6. When a new element B$ is encountered in line 50 it is compared in turn with all of the existing elements of A$( ). Thus, in 60, B$ is compared with the Lth element of A$( ). If it is greater than or equal to A$(L), then L is simply incremented in 70 and, if there are any elements of A$( ) left, the comparison is repeated. If B$ is less than A$(L), then it must be inserted. In order to make room for it, all the lower values of A$( ) must be moved down. This is done in 90-110 and the insertion is made in 120. If B$ is greater than all the members of the array, then it must be placed at the top of the list in the A$(N) position. To do this the entry is placed above the top of the list by setting L to N+1 in line 80 and then the whole of the array is moved down to make room for it. When all N of the entries have been inserted the assembled array is displayed on the screen.

Example 5.6

```
1 REM *** INSERTION SORT ***
10 INPUT"□NO.OF ENTRIES";N
20 IF N>10 THEN DIM A$(N)
30 FOR K=1 TO N
40 L=1
50 PRINT "■ENTRY NO."K;:INPUT B$
60 IF B$<A$(L) THEN 90
70 IF L<N THEN L=L+1:GOTO 60
80 L=N+1:REM B$> ALL A$(N)
90 FOR P=1 TO L-2
100 A$(P)=A$(P+1)
110 NEXT P
120 A$(L-1)=B$
130 NEXT K
140 PRINT"□"
150 FOR K=1 TO N
160 PRINT"■"A$(K)
170 NEXT K
180 END
```

5.4 SHELL SORT

As has been shown, a bubble sort rapidly gets out of hand when the number of elements to be sorted is large. However, if the data are not badly out of order initially, a bubble sort may be completed rapidly. A Shell sort makes several passes through the data before the final pass, which is in effect a bubble sort pass. Each pass orders the data to a higher degree. Whereas a bubble sort compares only adjacent elements, a Shell sort initially makes comparisons between elements which are far apart in the array on the assumption that the further apart the elements the more 'efficient' will be any corresponding interchange of data. With each successive pass the distance between the elements compared is halved until the final pass compares adjacent elements i.e. a bubble sort.

A typical Shell sort is shown in Example 5.7. The framework of this program is the same as in Example 5.2 so it is only necessary to consider the sort itself in lines 340-500. The sort interval V, which is the element separation over which the comparisons are made, is set up by initially setting V to equal N, the total number of elements and then re-setting V to INT(V/2). Thus, the initial sort interval is close to half the total number of elements. A total of (N-V) FOR . . . NEXT loops are then set up in 390 and in 420 the comparisons are made between elements separated by V. Any subsequent interchanges are made in 440-460 and flag Z is set to 1 before 480 initiates the next comparison. When all the comparisons have been made, if flag Z is set the comparisons are all repeated (line 490). This is because it is not possible to predict when a sort at a given sort interval will terminate. The process continues until line 490 finds Z=0 which shows that the elements are in order at the current sort interval. The sort interval must now be halved by returning to 360 but first the interval V is checked in 350, because if it is unity the sort is complete. If V is greater than 1 then it is halved in 360 and the sort repeated at the smaller sort interval. Thus, supposing N were 16, the initial value of V would be 8 to give an 8-sort, followed by a 4-sort, a 2-sort and finally a 1-sort, which is effectively a bubble sort. After the 1-sort is successfully completed the elements must be in order and the sort is exited from line 350.

Example 5.7

```
320 REM *** SHELL SORT ***
330 :
340 V=N:T1=TI
350 IF V<=1 THEN T2=TI: RETURN :REM SORT COMPLETE
360 V=INT(V/2):REM V=SORT INTERVAL
370 L=N-V
380 Z=0
390 FOR J=1 TO L
400 P=J+V
410 C=C+1:REM C=NO.OF COMPARISONS
420 IF A(J)<=A(P) THEN 480
430 E=E+1:REM E=NO.OF INTERCHANGES
440 W=A(J)    :REM INTERCHANGE DATA
450 A(J)=A(P):          :          :
460 A(P)=W    :          :          :
470 Z=1 :REM FLAG SET IF DATA NEEDS TO BE INTERCHANGED ON THIS PASS
480 NEXT J
490 IF Z>0 THEN 380:REM REPEAT THE SORT WITH SAME SORT INTERVAL
500 GOTO 350 :REM REPEAT THE SORT WITH A SMALLER SORT INTERVAL
```

The Shell sort is thus more complex than a bubble sort and for small values of N may even be slower. However, the sort effort for the Shell sort is proportional to $N^{1.5}$ compared to the N^2 of a

bubble sort with the result that for large values of N the Shell sort is considerably quicker. Thus, if N is set to 100 in the present program a typical analysis at the end of the sort would be:–

```
NUMBER OF COMPARISONS   =  2596
NUMBER OF INTERCHANGES  =   425
TIME TAKEN              =    77.4 SECONDS
```

which is typically twice as fast as the bubble sort.

5.5 QUICKSORT

Quicksort is the most complex of the techniques discussed here but it generally gives the fastest performance, particularly for large values of N. Its sort effort is proportional to NlogN which is superior to the sort effort of bubble and Shell sorts, for large numbers. In principle, Quicksort operates by splitting the main sort into a number of smaller sorts which in turn are split into even smaller sorts until the number of elements in the sub-groups is small enough for bubble sort to be efficient. One of the problems is that only one sub-sort can be executed at a time and so the location of the other sub-groups to be sorted must be 'memorized'. This is done by storing the start and end locations of the sub-groups in a special array called a 'stack'. The stack operates as a 'last-come, first-served' storage system. As each sub-group is sorted the boundaries of the next sub-group are 'pulled' off the top of the stack. When the stack is empty and all the sub-groups have been sorted the sort is complete.

The sort itself employs two pointers which are conventionally identified as I and J. At the start of the sort these are initialized to opposite ends of the complete array. Consider the array:–

```
45   61   14   8   85   54   23   39
```

Initially, pointer I would be associated with 45 and pointer J with 39. The element 45, being the left-most entry, is arbitrarily made the reference point of this part of the sort and is called the 'sort pivot'. The elements associated with pointers I and J are now compared and if necessary are interchanged. Thus in this case the array becomes:–

```
39   61   14   8   85   54   23   45
```

The sort pivot is still 45 but is now on the right and is associated with pointer J. The pointer opposite the sort pivot is now moved towards the sort pivot. Thus, in this case I moves from 39 to 61. The process is now repeated. The pointers therefore continually move towards one another and will eventually coincide. When this happens, the pointers mark the present position of the sort pivot and this is its correct position in the array. The array is now split into two sub-groups by the sort pivot.

The array now looks like;–

```
39   28   14   8   45   54   85   61
```

and it can be seen that the 45 is in its correct position. The positions of the 54 and the 61 are now pushed on to the stack so that the right-hand sub-group can be sorted later and then the left-hand sub-group is sorted by the same technique as above, 39 becoming the new sort pivot.

An example of Quicksort is shown in Example 5.8. Again the framework is the same as in Example 5.2 and the sort itself is situated in lines 340-500. The only modification to the main framework is in line 30 where the data array is, as usual, dimensioned to N and in addition the stack array S( ) is dimensioned to (20,2). Thus 30 DIM A(N), S(20,2):E=0:C=0. This is a compromise value. Clearly, if very large arrays are to be sorted these limits will have to be increased. If the dimension limit of S( ) is exceeded, then the program will terminate in 450 with a STACK OVERFLOW error message. In lines 340-350, the pointers I and J are initialised to the first and last elements of the array respectively. H is an indicator variable which takes a value of –1 when pointer I

indicates the sort pivot and a value of +1 when pointer J indicates the sort pivot. The comparison of elements is made in 370. If no interchange is needed, i.e. if A(I) is less than or equal to A(J) then a jump is made to 410 where either I is incremented or J is decremented, depending on the sign of H which indicates the position of the sort pivot. If an interchange is required then a simple bubble sort switch is made in 390 followed by a change in the sign of H in 400 and again a change in the pointers I and J. The pointers are then compared in 430. If I is less than J, then the pointers have not yet coincided and the sort continues within the same group. If the pointers coincide then, if I+1 is not greater than J1, i.e. if the (correctly positioned) sort pivot is not the largest number in the array, then it will be necessary to push the boundary pointers of the upper sub-group on to the stack. This is done by incrementing the stack counter P and then feeding I+1 and J1 on to the stack in 460. The sub-array boundaries are then set to those of the lower array in 470 and the sort resumed. When all the lower sub-groups have been sorted the stack counter is tested in 480. If it is not zero then the most recent upper sub-group is pulled off the stack (line 490) and sorted. This continues until the stack is empty (P=0) when the sort routine is exited from line 480.

Example 5.8

```
320 REM *** QUICK SORT ***
330 :
340 I1=1:J1=N:T1=TI
350 I=I1:J=J1:H=-1:REM SET POINTERS
360 C=C+1:REM C=NO.OF COMPARISONS
370 IF A(I)<=A(J) THEN 410
380 E=E+1:REM E=NO.OF INTERCHANGES
390 W=A(I):A(I)=A(J):A(J)=W
400 H=SGN(-H)
410 IF H=1 THEN I=I+1   :GOTO 430
420 J=J-1
430 IF I<J THEN360
440 IFI+1>J1 THEN 470
450 P=P+1:IF P>20 THEN PRINT"STACK OVERFLOW":END
460 S(P,1)=I+1:S(P,2)=J1:REM PUSH ONTO STACK
470 J1=I-1:IF I1<J1 THEN 350
480 IF P=0 THEN T2=TI:RETURN
490 I1=S(P,1):J1=S(P,2):P=P-1:REM PULL OFF STACK
500 GOTO 350
```

The Quicksort usually gives very satisfactory results, as can be shown by setting N to 100 in this program. A typical analysis at the end of the sort would be:–

NUMBER OF COMPARISONS	=	667
NUMBER OF INTERCHANGES	=	209
TIME TAKEN	=	32.8 SECONDS

This is typically twice as fast as the Shell sort and possibly four times faster than the bubble sort, assuming randomly presented data. However, when N is small the extra complexity of the Quicksort can be a disadvantage and the sort may be slower than other less complicated techniques.

CHAPTER 6

Good Programming

6 Good Programming

It is relatively easy to learn to write working programs in BASIC. However, the nature of the language is such that it is also very easy to write extremely bad programs, which are unintelligible to anyone reading the listing, and very difficult to modify.

Some purists would argue that it is impossible to write really good programs in BASIC, and indeed the lack of certain facilities available in other high-level programming languages makes the writing of long BASIC programs – several hundred lines or more – very difficult. Nevertheless, if a few simple principles are applied, it is possible to produce reasonably well-constructed programs which are reasonably intelligible and easy to modify.

The most important requirement is that the program should be well-designed – i.e., the flow of ideas through the program should be simple, straightforward, and easy to follow. This is often quite difficult to achieve but is worth a fair amount of effort, since some time devoted to thinking about the problem at the design stage may save a great deal of time and trouble later.

6.1 PROGRAM PLANNING

Probably the greatest pitfall awaiting the BASIC programmer is the fact that it is extremely easy to go to the machine and start typing in the program directly. For any but the simplest programs however this is fatal, and almost guarantees that the resulting program will be badly-written.

NEVER, NEVER, NEVER start to write a program at the machine. Get well away from the machine, with a paper and pencil (and a rubber) and THINK. Think about the problem. What is it you have to do? What are the pitfalls? How can you best approach the problem? If the problem is complex you will probably find it helpful to draw a flow-chart at this point, breaking down into individual steps the process your program is designed to carry out. When you have done this, write out your program on paper. It is at this stage that you should try to anticipate problems – out-of-range data entries, blank returns, wrong data types, negative numbers instead of positive, and so on.

6.2 BLOCK STRUCTURE

So far as possible break up the program into separate blocks, each of which is complete in itself. Minimise the number of GOTO statements. Make sure that there is only one point of entry to each block, and only one exit. This helps to clarify the operation of the program and makes it easier to modify, should this subsequently be necessary. It also reduces the likelihood that your program will behave in unexpected ways.

6.3 MAKING YOUR PROGRAMS INTELLIGIBLE

Make your program INTELLIGIBLE, both for yourself and others. You will be surprised at the way in which you forget the working even of the simplest programs after a few weeks or months. Don't be afraid of REM statements. Comment your program fully – some programmers write the comments first, to act as a skeleton around which the flesh of the program can be built.

Don't forget to give every program a title – one listing looks very much like another after a few months in your desk drawer. If the program is to be used by others, put your name on it, so that they

know where to come for information and advice (or with complaints!). In particular, include in the title of the program a version number, or, at the very least, the date, so that you can tell subsequently which is the most up-to-date version of the many listings you will undoubtedly accumulate.

An introduction to explain the purpose and functioning of the program is often helpful. In the case of a large program, this introduction may usefully contain a table of variable names, explaining their significance, as well as definitions of functions, and DIM statements. In general, any item of program-wide significance should be placed at the head or at the end of the program.

Lay out your program as clearly as possible. Leave spaces between words, and in general put each statement on a separate line, except where several short statements form a well-defined group – for instance, a simple delay routine.

Use blank REM lines to separate blocks of programs. (A similar effect can be obtained by using a colon instead of REM.) If you're really fussy, make sure that all your line numbers are the same length, as this looks much prettier.

Subroutines should be gathered together at the end of the program, and given high, memorable line numbers. Label your subroutines, and separate them by blank REM statements. Don't forget to put an END statement before your subroutines.

Basic variable names are unhelpful, being effectively limited to two characters. PET variable names up to sixteen characters in length are allowed, but this is, in fact, a dangerous snare. The PET examines only the first two letters of the name, so that apparently distinct variable names such as, for instance, "DEPTH" and "DECAYTIME", would not be distinguished by the PET. It is therefore probably safer to stick to two-character variable names. So far as possible these should suggest the quantity represented by the variable – e.g. DE for delay, T1 and T2 for initial and final values of time, and so on.

6.4 TESTING

When you have written and entered your program, TEST and validate it thoroughly. Wherever possible, the program should be tested in sections; the tested blocks can then be linked into larger groupings which are tested in their turn, and so on. It is often helpful to build in small testing and debugging routines, which print out the values of variables at intermediate stages in the execution of the program. These can be removed when testing is complete. Alternatively, especially in a large program, it may be convenient to leave them permanently in place, "switching" them off or on as required by means of a software flag. When you are satisfied that you have tested the program thoroughly, get someone else to test it – he will certainly find some bugs that you have missed.

Finally, if a program is large and complex, REM statements may not be adequate to explain its workings, so that additional documentation is required. Write such documentation immediately the program is complete; if you delay, it is possible that the documentation will never be written and, even if it is eventually written, you will have forgotten many important points.

CHAPTER 7

Using PET/CBM Peripherals – Logical Files

7 Using PET/CBM Peripherals – Logical Files

PET/CBM PERIPHERALS

By "peripherals" we normally mean devices such as the printer, disk drive unit, or even the cassette recorder, which are external to the computer itself. In the PET system, however, even such basic input/output devices as the screen and keyboard can also, if desired, be handled in the same manner as these external devices. We shall see in Chapter 12 that this facility can be used to good advantage. However, to simplify things in normal operation, the designers of the PET have provided additional routines to make the use of the screen and keyboard, and the loading and saving of programs on the cassette, less complicated.

7.1 Logical files

Except in a few cases such as the screen, keyboard and, to some extent, the cassette, the programmer must communicate with each peripheral by means of a "logical file". This technique has a number of advantages, and is widely used in many different computer systems of all sizes, but unfortunately it is rather difficult to grasp when first encountered.

Information (programs or data) is stored by the PET on tape or disk in the form of *files*. The principle is just the same as that of the familiar office file, and a number of commands are available in PET BASIC to permit information to be written into or read from such "logical files". These commands are also pressed into use to allow data to be written to or read from peripherals, which are in consequence treated for programming purposes exactly as if they were logical files containing stored information.

Several logical files may be in use simultaneously (up to ten at a time in the PET), and each is therefore labelled with a number so that they can be distinguished. This "logical file number" can have any value from 1 to 255, and can be selected at will within this range by the programmer. However, we shall see from the practical examples that it is usual to stick to low numbers and, where a peripheral is concerned, to select numbers related to the characteristics of the specific device addressed, as this makes it easier for the programmer to keep track of what is happening.

The instruction by which we open a file to a peripheral contains information about that peripheral, so that once we have opened a file within a program, we can thereafter address the particular peripheral concerned by means of the appropriate logical file number, until the file is closed again (normally at the end of the program).

When dealing with a peripheral, the information contained in the statement which opens a logical file consists essentially of two items, although we shall see in later chapters that where the logical file is a data storage file, the opening command can in some cases contain quite a lot of additional information. These two basic items are termed "addresses" – the "primary address" or "device number" and the "secondary address".

7.2 PRIMARY ADDRESS – DEVICE NUMBER

Most PET peripherals are electrically connected to the PET itself by a system termed the "IEEE-488

bus". (The "IEEE" indicates that the system conforms to a standard set by the American Institution of Electrical and Electronic Engineers.)

This system is rather like the ring-main used in wiring a house – all the peripherals are connected in parallel to the same communicating wires, just as the domestic TV, washing machine, iron, etc are all connected in parallel to the house wiring. In the house, we turn on any specific item of equipment by operating a switch; in our computer system we obtain the same effect by sending out a signal – the primary address – which *tells* the peripherals which of them is being addressed. We may compare this with a school-teacher addressing a class. If she simply says "stand up" or "come here", the class will not know to which of them the instruction is addressed. However, if she says, for instance, "Clarence Cholmondeley, come here", there is no problem. In the PET system, the primary addresses are just numbers. Thus, the primary address of the PET printer is 4, that of the disk drive unit is 8, and so on. These numbers are set during manufacture, but can, if necessary, be changed, so that we can use more than one printer or disk drive unit on the same system.

The standard primary addresses or device numbers in the PET system are:

0 keyboard.

1 console cassette unit where fitted, or the external cassette unit (rear or side port) if the PET is one of those models not fitted with a console cassette unit.

2 second cassette unit – if a console cassette is fitted a second cassette unit can be plugged into the rear port. In machines with no console cassette unit, the second cassette unit can be connected to an additional port.

3 screen.

4 printer.

8 disk drive unit.

Other numbers up to 30 may be used for additional peripherals connected to the IEEE bus. This is particularly useful in technical applications where instruments such as digital voltmeters, electronic thermometers, etc., may be controlled by the PET and feed information back to it via the bus.

If no primary address is specified by the programmer, the PET assumes a "default value" of 1 so that if the primary address of the peripheral is not given, the PET automatically assumes that the cassette drive is required.

7.3 SECONDARY ADDRESS

This is not always needed. The secondary address is essentially an *instruction* to the peripheral specified by the primary address. Some peripherals such as the PET printer, or the disk drive unit, are "intelligent" – i.e. they contain a microprocessor, and are capable of carrying out quite complicated operations without detailed instructions from the PET. The secondary address simply tells the peripheral which item from its repertoire is required. This facility is dealt with at greater length in the chapters dealing with the printer and disk drive unit – Chapters 8 and 9.

Where a secondary address is required, but the user fails to specify a value, the PET will normally assume a default value of zero. The precise significance of a specific secondary address number will depend on the peripheral which is being addressed.

7.4 OPENING AND CLOSING A LOGICAL FILE

Now that we have established what the logical file must contain, we can go ahead and actually open a

file in the PET. This done by means of the OPEN command. Where peripherals are concerned this has the basic form

OPENlfn,pa,sa

where lfn is the logical file number, with a value in the range 1 – 255
pa is the primary address, with a value in the range 1 – 30
sa is the secondary address, with a value in the range 2 – 14

The command

OPEN2,4

will thus open a channel to the printer; this channel can then subsequently be referred to by the logical file number – in this case 2. The (assumed) secondary address is 0.

Since there is a limit to the number of files which can be open simultaneously, we must CLOSE the file when it is no longer required, and certainly before the program ENDs. The CLOSE command is very simple. It has the form

CLOSElfn

Thus the logical file OPENed above would be CLOSEd by the command

CLOSE2

Both these commands can be used either within a program or in direct mode.

If by mistake a second attempt is made to OPEN a file which has not been closed in this way, an error message – ? FILE OPEN ERROR – will be displayed, and the file will automatically be CLOSED by the PET. The file must then, rather confusingly, be OPENed once again before it can be used.

7.5 SENDING INFORMATION TO A PERIPHERAL – PRINT#

Once we have opened a channel to a specific peripheral, we subsequently address that peripheral by means of the logical file number. We send information to a peripheral by means of the PRINT# command.

Thus the sequence

```
OPEN2,4
PRINT#2, "HELLO"
CLOSE2
```

would cause a channel (lfn = 2) to be opened to the printer, the word "HELLO" to be printed, and the channel closed. (No secondary address is used, since the special facilities of the intelligent printer are not required.)

The PRINT# command is used whenever information is to be sent to any peripheral connected to the IEEE bus, whether this be the printer, the disk drive unit, or some other device such as a digital voltmeter, electronic thermometer or counter/timer.

NOTE that the PRINT# command must be typed out *in full*, and that no spaces must be left between the characters. PRINT# *must not* be input using the "?" symbol – this will give an apparently correct program listing, but will not work.

7.6 GETTING INFORMATION FROM A PERIPHERAL – INPUT# AND GET#

Once a logical file has been opened, we can request information from the peripheral using either the INPUT# or the GET# command. The GET# command inputs a single character from the

peripheral, just as the familiar GET command accepts a single character from the keyboard. The GET# command is rather specialised in its applications however, and is likely to be used only by the more expert programmer.

The INPUT# command also functions in a very similar way to the familiar INPUT statement, reading a string or the value of a variable from the peripheral. Several strings or values may be input using a single INPUT# command.

Thus, for instance, if we have opened a communication channel with the disk drive, using the logical file number 2, say, we can read in items of data from the disk very simply.

```
INPUT#2,N
```

will cause a number to be read from the disk and assigned as the value of the variable N.

```
INPUT#2,A$
```

will cause a string to be read from disk, and assigned to the string variable A$.

Several values may be read in at once, and these may be a mixture of strings and numerical variables.

For example, the command

```
INPUT#2,N$,N,A$,B$,C
```

is perfectly valid.

7.7 SUMMARY

We have seen that in general the PET communicates with its peripherals by means of *logical files*. The statement which opens a file contains a *primary address* or device number and (optionally) a *secondary address* which in the case of an intelligent peripheral invokes some specific specialised function. Different logical files are distinguished by labelling them with separate *logical file numbers*.

A logical file must be OPENed before use and CLOSEd before leaving the program. Information is sent to peripherals using the PRINT# statement, and read from peripherals by means of the INPUT# or (less commonly) the GET# statement.

Chapters 8 and 9 contain many examples of the use of logical files to communicate with the printer and disk drive, and even with the keyboard. A logical file must also be used when data, as distinct from program material, is to be stored on or read from cassette.

CHAPTER 8

The PET/CBM Printer

8 The PET/CBM Printer

INTRODUCTION

The PET printer is "intelligent" – that is, unlike most other microcomputer printers, it contains a separate microprocessor of its own. This makes possible a number of very useful features not available on most printers. Of course, to make use of these extra facilities involves some added complexity in your programs but, once mastered, such features as the ability to control the print format and line spacing, or to advance the printer to a new page, all under program control, will be found invaluable. The formatting facility is especially useful in commercial and technical applications.

8.1 THE PRINTER AND LOGICAL FILES

Like all those PET peripherals which use the IEEE-488 bus, the printer is handled as if it were a logical file (see Chapter 7). Thus, once a logical file has been OPENed which corresponds to the printer, the PET sends text and instructions to the printer by means of PRINT# commands, just as it would do to a data file (see Chapters 7, 10 and 11.)

With the printer the most basic form of the OPEN command is used. This basic format is simply

OPENlfn,pa,sa

lfn is the logical file number, which as usual may be selected at will in the range 1-255.

pa is the primary address or device number of the printer. For the printer this is set during manufacture to 4, although it may be altered by an engineer. This may be necessary for example if two printers are to be used in conjunction with a single machine (other CBM primary addresses are listed in section 7.2).

In the OPEN statement, sa is the secondary address which may have any value from 0 to 7 for 2000- and 3000- series printers, or 0 to 10 for other models. Where no secondary address is specified, the "default value" – i.e. the value assumed by the PET – is 0. This causes all text to be printed just as received, without using any of the special facilities of the PET printer. The use of the secondary address is dealt with in the later parts of this chapter.

As always, it is convenient to choose a logical file number which is the same as the primary or secondary address, so that it is more readily identified where it occurs in the body of a program.

8.2 THE PRINT# COMMAND

Once a logical file corresponding to the printer has been opened within the PET, text or data may be output to the printer using the command

PRINT#lfn,data

(Remember that the file must be closed before leaving the program.)

For example, the sequence

```
OPEN 1,4
PRINT#1, "HELLO"
CLOSE 1
```

will cause a logical file (number 1) corresponding to the printer to be opened, the text string HELLO to be printed and the file to be closed. The command PRINT#lfn opens the physical connection to the printer and then closes it when the requisite text has been transmitted.

The logical file number specified in any PRINT# command must of course be that of a file which has already been OPENed.

N.B. *PRINT# must be typed in full; the abbreviation ? # is not permissible, and no spaces must be left between the characters of the command.*

Similarly, the sequence

```
OPEN 1,4
PRINT#1, X
PRINT#1,A$
CLOSE1
```

will cause the current values of the variable X and the string variable A$ to be printed.

These features may be seen from the following simple demonstration program.

```
10 REM PRINTING STRINGS AND VARIABLES
20 REM
30 PRINT "WHAT IS YOUR NAME":INPUT N$
40 PRINT "HOW OLD ARE YOU":INPUT A
50 OPEN4,4:REM  OPEN LOGICAL FILE CORRESPONDING TO PRINTER
60 PRINT#4,N$" IS "A
70 CLOSE4
80 END
```

After OPENing a file to the printer (line 50), line 60 causes the current value of the string variable N$, the string " IS " and the current value of A to be printed, so:

```
JOHN SMITH IS 86
```

8.3 THE CMD COMMAND

The CMD command takes the form

```
CMDlfn
```

and effectively transfers control from the computer to the printer. Again, the logical file number must be the same as in the appropriate OPEN statement.

Subsequent output is then directed to the printer rather than to the screen.

This is useful where a program or a disk directory is to be LISTed; text or data may also be printed using the simple PRINT or ? command normally used to output text or data to the screen. Since control is now in the hands of the printer, the communication channel between computer and printer remains open, *even if the corresponding logical file in the computer is closed*. The CMD command must therefore be followed by a PRINT# statement to break the communication link (i.e. to "unlisten" the printer) before the file is closed. The CMD command thus allows several peripherals to use the bus simultaneously. (As, for example, when listing a disk directory; both the disk drive unit and the printer

must be able to communicate with the computer during this process, unless the directory has previously been loaded into memory.)

Thus, we might write

```
10 OPEN4,4
20 PRINT#4, "DEMONSTRATION"
30 CLOSE4
```

This would cause the single word "DEMONSTRATION" to be printed.

Alternatively, we might use the sequence

```
10 OPEN 17,4
20 CMD 17
30 PRINT "DEMONSTRATION"
40 PRINT#17
50 CLOSE 17
```

If a faulty instruction is input – e.g. if we attempt to open a file which has already been opened – the computer automatically closes *all* the logical files which are open at that time. It does not, however, "unlisten" the peripherals, so that the files should be re-opened and a PRINT# command given for each file before closing them again.

8.4 LISTING A PROGRAM

To obtain a printed listing of a BASIC program present in machine memory the following instruction sequence may be used:

```
OPEN 2,4
CMD 2
LIST
PRINT#2
CLOSE 2
```

If the LIST instruction is used *within a program,* it automatically terminates the program; it must therefore be the last statement of the program. The commands

```
PRINT#lfn
```

and

```
CLOSElfn
```

should therefore be input from the keyboard after the termination of the program.

Try using the above commands, first inputting them in direct mode from the keyboard and then incorporating them within simple programs. When using direct mode, it is more convenient to type the first three commands on a single line:

```
OPEN2,4:CMD2:LIST
```

A similar instruction sequence may be used to print a disk directory – this is especially simple when UNIVERSAL WEDGE or DOS SUPPORT is used.

8.5 UPPER AND LOWER CASE

Although the printer normally operates in upper case, it may be switched from upper to lower case, and vice versa; furthermore, both upper and lower case may be used on the same line.

The case may be controlled by means of the "cursor up" and "cursor down" controls on the keyboard. For convenience these signals will be represented here by <CU> and <CD>. If the "cursor down" signal is transmitted as part of a string (i.e. a piece of text), then all the remaining characters on the same line subsequent to the <CD> will be printed in lower case; the graphics characters will however be unavailable. If a <CU> is sent later in the same line, subsequent characters will be printed in upper case (and the graphics characters will again be available). In any case, as soon as an instruction to the printer is terminated by transmitting a line feed (RETURN), the printer will automatically revert to operation in upper case. The sequence

```
OPEN4, 4
PRINT#4, "<CD>LOWER CASE"
CLOSE4
```

will thus cause to be printed the following:

```
lower case
```

The sequence

```
OPEN3,4
PRINT#3,"<CD>LOWER CASE<CU>UPPER CASE"
CLOSE3
```

will produce

```
lower case UPPER CASE
```

8.6 CHARACTER CODES – THE CHR$ FUNCTION

All characters are transmitted from the computer to the peripherals as numbers. We may thus replace "<CU>" and "<CD>" by the equivalent numerical codes – CHR$(145) and CHRS(17) respectively.

The above sequences would then become

```
OPEN4,4
PRINT#4,CHR$(17)"LOWER CASE"
CLOSE4
```

and

```
OPEN3,4
PRINT#3, CHR$(17)"LOWER CASE"CHR$(145)"UPPER CASE"
CLOSE3
```

8.6.1 Other uses of the CHR$ function

This method of transmitting characters to the printer is not essential where the <CU> and <CD> signals are concerned, but *some other characters must be sent in this form.*

For example, if inverted commas are to be printed out within a string of text, they cannot be sent in the normal way – e.g. "SAY "HELLO" " – since the second quotation mark will be taken to indicate the end of the string. In this case the inner quotation marks must be input as the appropriate ASCII code – CHR$(34):

```
OPEN200,4
PRINT#200, "SAY" CHR$(34)"HELLO" CHR$(34)
CLOSE200
```

Similarly, when using disk drives, it may sometimes be necessary to transmit in this form characters commonly used as terminators – the colon, RETURN, etc.

NB Where some CHR$ expression is to be used repeatedly within a program, it may be preferable to define a string variable with the appropriate value. For example, the last instruction sequence would then become:

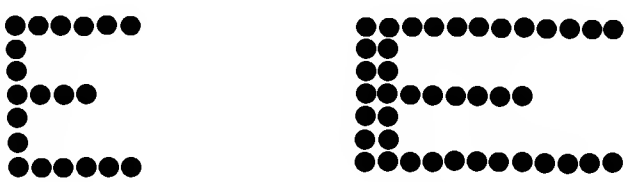
```
OPEN200,4
C$ = CHR$(34)
PRINT#200, "SAY"C$"HELLO"C$
CLOSE200
```

8.7 MORE ADVANCED APPLICATIONS OF THE CBM PRINTER

8.7.1 Enhanced characters

The PET printer is a "matrix" printer; each character is formed by an array or matrix of separate dots. Each character is built up in this way within a rectangular grid. The grid has 6 vertical lines of dots and seven horizontal rows, so that a completely black rectangle (such as a reverse-contrast space) will be represented by a complete matrix of 42 individual dots.

Any character may, for emphasis, be printed twice its normal width; each column of the 7-row 6-column matrix is then repeated, so that an 'enhanced' character occupies a 7-row 12-column matrix.



Normal Enhanced

This is achieved by sending the control signal CHR$(1) in front of the character or characters to be enhanced. The character format is returned to normal by the control signal CHR$(129). These characters have no equivalent on the keyboard and so must be sent in this manner. The format is automatically returned to normal when an instruction to the printer is terminated by the transmission of a line feed (RETURN). Thus, if a number of lines are to be printed in enhanced characters, the character CHR$(1) must be sent at the start of every line.

Enhanced characters are useful in headings, or to emphasise selected words or letters.

For example:

```
OPEN1,4
PRINT#1, CHR$(1)"H"CHR$(129)"ELLO"
CLOSE1
```

will cause the word "HELLO" to be printed, with the first letter H enhanced.

8.7.2 Multiple enhancement

A character may be enhanced more than once, so as to occupy a matrix 18, 24, 30, or even more columns in width; for example, the instruction sequence.

```
OPEN4,4
PRINT#4, "S"CHR$(1)"A"CHR$(1)"L"CHR$(1)"F"CHR$(1)"O"CHR$(1)"R"CHR$(1)"D"
CLOSE4
```

will cause each successive letter to be enhanced by an additional 6 columns.

8.7.3 Paging

Normally, the printer prints continuously, putting 66 lines of text on each standard 11-inch page. It is, however, possible to make the printer operate in a "paging" mode.

When this option is selected, the printer prints 60 lines of text per standard page, with three blank lines at the top and three at the bottom of each page (assuming of course that the paper is initially correctly set).

Once the printer has been set to the paging mode it will continue to operate in this mode until either paging is turned off again, or the machine itself is switched off.

"Paging" is turned on by transmitting the character CHR$(147) and turned off by transmitting CHR$(19). CHR$(19) represents the character generated by pressing the HOME key of the keyboard while CHR$(147) represents CLR, generated by pressing SHIFT and HOME simultaneously.

For example, the sequence of instructions

```
OPEN4,4
PRINT#4,CHR$(147)
PRINT#4,data
PRINT#4,CHR$(19)
CLOSE4
```

will turn on the paging facility, print out the data or text, and turn off paging.

8.7.4 The head-of-form facility

When paging is turned off in this way, the printer will advance to the head of the next page, ready to start printing again. The paging-off command thus offers a "head-of-form" facility, allowing text to be printed repetitively on successive pages, in the same position on each page.

The paging instruction may if preferred be transmitted as a string, using the HOME and CLR (shifted HOME) keys.

```
PRINT#4,"<CLR>"
```

and

```
PRINT#4,"<HOME>"
```

would thus turn paging on and off, respectively.

The use of the head-of-form facility is demonstrated by the following simple program:

```
10 OPEN4,4
20 PRINT#4,CHR$(147)
30 PRINT#4,TAB(12)CHR$(1)"TOP OF FIRST FORM"
40 PRINT#4
50 PRINT#4, "THIS PROGRAM DEMONSTRATES THE USE OF HEAD-
           OF-FORM FACILITY."
60 PRINT#4,CHR$(19)
70 PRINT#4,TAB(12)CHR$(1)"TOP OF NEXT FORM"
80 CLOSE 4
90 END
```

8.7.5 Overprinting

It is possible to overprint a line by forcing a carriage return without a line feed. This is done by transmitting the character CHR$(141). The facility is useful in synthesising symbols or characters – such as the £ sign – not present on the PET keyboard. It is also invaluable where a heading or title is to be underlined.

The following short program demonstrates how a reasonable £ sign may be printed.

```
10 OPEN6,4
20 PRINT#6,CHR$(17)"F100"CHR$(141)"_"
30 CLOSE
```

The CHR$(17) sets the printer to lower case, so that 'F' in the immediately following string is printed as 'f'. A carriage return without line feed is then carried out, and the 'f' is underlined, so producing a recognisable £ sign: £. (The underline is produced by simultaneous depression of the SHIFT and $ keys.)

8.8 THE SECONDARY ADDRESS

All the facilities described so far have been invoked by the use of control characters, the assumed value of the secondary address being zero. There are, however, a number of other invaluable facilities which may be called into operation by means of the so far unused secondary address in the OPEN statement. In the case of the CBM 2022 and 3022 (tractor feed) models, seven different facilities may be called into play by using the appropriate secondary addresses. Most, but not all, of thesc facilities are also available on the corresponding friction-feed models. Eleven secondary addresses are available on the 4022 printer.

Note that in some cases several OPEN statements in succession may be necessary, to set all the required options. (Up to ten files may be open simultaneously.)

The seven available secondary addresses available on all the tractor-feed printers are:

0 Print data as received. (This is the default value – i.e. the value assumed by the printer if no secondary address is included in the OPEN statement.)

1 Print data according to a previously specified format. (The format is separately specified using an OPEN command with secondary address 2.)

2 Accept and store formatting data

3 Set page size (for use with "head-of-form" facility)

4 Enable diagnostic (error) messages to be output by the printer

5 Define a special character, of form specified by programmer

6 Adjust spacing between lines (tractor-feed models only).

8.8.1 Formatting – Secondary Addresses 1 and 2

We have already seen that if no secondary address (or a secondary address of 0) is used in the OPEN statement, text and data will be printed exactly as they are received from the PET by the printer. This can be inconvenient, as it may be difficult to arrange a string precisely as required; also, unless we take special precautions, the value of a numerical variable will be printed using the maximum available number of significant figures, just as it would be on the screen.

These difficulties can be avoided by making use of the automatic formatting facilities of the PET printer. This is done by sending to the printer a string which specifies the desired format. This is stored in the printer, and whenever formatted output is specified by the program, the data sent are printed exactly according to the specified format.

NOTE however that there are a number of 'bugs' in the PET printer firmware. If something goes wrong when you use the more advanced facilities, do not automatically assume that it is your fault. These bugs are particularly irksome in the case of the formatting facility. Do not be put off by them – the facility is so useful that it is worth persevering.

8.8.2 Specifying the format

The format is sent to the printer as a single string, using dummy symbols for letters and numbers. This format string is sent via a logical file with a secondary address of 2. Subsequently, text and data will be printed according to the specified format if they are sent to the printer by means of a logical file with a secondary address of 1.

We can mix string variables (i.e. alphanumeric symbols) and numerical variables on the same line. However, the formats for the two types of variable are differently specified.

Alpha-numeric Strings

The dummy character used within the format statement is "A"; blanks are indicated simply by blanks, so:

```
AAAAA  AA  AAAAAAA
```

Each string variable sent within a single print statement is allocated to the next group of A's in the format statement in turn, one character being printed for each "A". Excess characters, and leading

blanks, are discarded. Thus, the following instruction sequences (which could be input in either direct or program mode) would produce varying results according to the length of the string variable Z$.

```
10 OPEN 1,4,1
20 OPEN 2,4,2
30 PRINT#2, "AAAAA"
40 PRINT#1,Z$
50 CLOSE1:CLOSE2
60 END
```

If the string represented by Z$ has five letters, it will fit exactly into the specified format; if more than five, it will be shortened to fit. Thus, if we run the program with Z$ equal in turn to, say, "PETER", "MARMADUKE" and "TOM", we shall get

```
PETER
MARMA
TOM
```

We can format several string variables on the same line, but in this case we *must* follow every string, apart from the last, with the "skip" character CHR$(29), to indicate the end of that string. Ordinary delimiters such as the colon, full stop, comma, etc will be treated as part of the string; *only the special skip character will be recognised.* The skip character is not necessary after numerical variables, however. The skip character CHR$(29) is the signal produced by pressing the CURSOR RIGHT (→) key on the keyboard, so that it may be sent either as CHR$(29) or as "→".

If one string is to consist solely of blanks, this too must be indicated by a special character. Otherwise, as we have already seen, leading blanks are ignored, and the printer will therefore ignore the whole of the blank string. The null string character is CHR$(160).

The following short program demonstrates these features.

```
10 OPEN 1,4,1
20 OPEN 2,4,2
30 PRINT#2,"AAAAA . . AAA . . . . . AAAAAA . . AAAAAAA"
40 Z$="COMPUTER":Y$="CALCULATOR"
50 PRINT#1, Z$CHR$(29)Y$CHR$(29)CHR$(160)CHR$(29)Z$
60 CLOSE1:CLOSE2
70 END
```

This will produce the following print-out:

```
COMPU   CAL                COMPUTE
```

The effect of the null string may clearly be seen. This process is fairly complicated and some experimentation may be necessary before a program produces the desired results.

Numerical variables

The dummy character used in the case of numerical variables may be either Z or 9; the two produce quite different results. The decimal point, if any, is indicated in the usual way.

Where 9 is used as the dummy digit, leading zeros (before the decimal point) are suppressed and replaced by blanks. If Z is used (and this is allowed only *before* the decimal point) leading zeros are printed, and indeed inserted if necessary to fill out the format.

Possible formats thus include, for example,

99.999 ZZZ.99 ZZ 99

Consider the two similar formats ZZZ.99 and 999.99. If now the number .123 is printed using these two formats in turn, the results will be respectively:

000.12 and .12.

In each case, extra trailing digits are ignored.

Serious errors could result in the case of numbers too large to fit within the specified format; to avoid confusion in such a case, the digits are not printed at all, but replaced by asterisks (*). Thus, if we try to print, for instance, 12345.67, using the format 999.99, the print-out will be

*** **

Thus we can see at a glance whenever such an overflow has occurred.

Sign

The sign of a number will not be printed unless this is specified in the format statement. There are two possible forms. If the letter S *precedes* the string then the sign, whether + or –, will always be printed *in the specified column position*. If however the format string is *terminated* by a – sign, then a positive number will be followed by a blank and a negative number by a minus sign.

As with strings, several numbers may be printed on the same line. For example:

```
10 OPEN1,4,1:OPEN2,4,2
20 PRINT#2,"SZZZ.99 SZZZ ZZ.99- 999"
30 X=-123.456: Y=6.789
40 PRINT#1,X,X,X,X
50 PRINT#1,Y,Y,Y,Y
60 CLOSE1:CLOSE2
70 END
```

The print-out from this program will help to make the effects of the various format options more clear.

Mixed Strings

In general, we will need to mix both numerical variables and strings on the same line. This is readily done by means of a combination of the various dummy symbols discussed. However, care must be taken to insert all the necessary skip characters if the formatting is to be successful.

At this stage two bugs should be noted.

1 When a formatted message has been printed, an attempt to send a second format string to the printer immediately will lead to unexpected and unwanted results. This problem may be avoided by printing an *unformatted* message before attempting to input a new format. A blank line, or even a carriage return without line feed [CHR$(141)] will do. The need for this

may be confirmed by removing line 280 in the demonstration program below, and running the program several times in succession.

2 Occasionally it may be impossible to obtain satisfactory results with some particular format. In this case it will usually be found that simply re-arranging the order of the variables in the format string will produce satisfactory results.

Example 8.1

The following demonstration program gives some idea of the use of this facility. It produces a formatted print-out in the form of an imitation stock-list, as shown below the listing. Lines 130, 280 and 310 are present solely to avoid the first of the two bugs just described. The skip character is allocated a string variable name – C$ – for convenience. The strings "ITEM" and "WIDGETS, PURPLE, TYPE" are also allocated to string variables. This will generally be found convenient since it simplifies any changes in the text which may subsequently be required. It is convenient also to transmit the format string itself as a variable; among other things, this has the advantage that a longer format string can be used than when this is input directly as part of the PRINT# statement. If you really want to use every one of the printer's 80 characters per line, it will be necessary to build up the string using two separate string variables, since otherwise it is impossible to input an 80-character string.

```
100 REM       MIXED FORMAT STRINGS
110 REM       ***** ****** *******
120 REM
130 OPEN4,4
140 OPEN1,4,1
150 OPEN2,4,2
160 F$="AAAA 99          AAAAAAAAAAAAAAAAAAAA ZZ       9999"
170 REM       TRANSMIT FORMAT STRING
180 PRINT#2,F$
190 A$="WIDGETS,PURPLE,TYPE"
200 B$="ITEM"
210 B=56
220 C$=CHR$(29)
230 FOR I=1 TO 5
240 A=INT(RND(1)*100)
250 C=20+I
260 PRINT#1,B$,C$,I,A$,C$,C,A
270 NEXT I
280 PRINT#4
290 CLOSE1
300 CLOSE2
310 CLOSE4
320 END
```

```
ITEM  1         WIDGETS,PURPLE,TYPE 21         71
ITEM  2         WIDGETS,PURPLE,TYPE 22         85
ITEM  3         WIDGETS,PURPLE,TYPE 23         15
ITEM  4         WIDGETS,PURPLE,TYPE 24         67
ITEM  5         WIDGETS,PURPLE,TYPE 25         22
```

8.8.3 Setting the number of lines per page – secondary address 3

This option is used in conjunction with the head-of-form facility where paper with pages of length

other than the standard eleven inches is being used, or where it is desired to print text repetitively at fixed intervals, several times per page. (Tractor feed models only).

The instruction sequence necessary to change the page length differs slightly for the 3000- and 4000-series printers. With a 3022 printer the sequence would be of the form

```
OPEN6,4,3
PRINT#6,15
CLOSE6
```

With a 4022 printer, however, the second command would be

```
PRINT#6,CHR$(15)
```

The number transmitted in the PRINT# statement determines the effective page length. Some experiment may be required before the desired page length is obtained. Once the page length has been set to a non-standard value in this way, it will remain at the new value until reset, or until the printer is turned off.

8.8.4 Error messages – secondary address 4

The printer can issue a number of simple diagnostic error messages. If these are required, a file with a secondary address of 4 must be opened (and later closed, of course).

For example:

```
OPEN4,4,4:REM ENABLE ERROR MESSAGES
        )
        )
        )  (body of program)
        )
        )
CLOSE4
```

8.8.5 Special characters – secondary address 5

The user can define any character which can be accommodated in the standard 7 × 6 matrix. The characters are specified by means of a series of decimal numbers each specifying the contents of a column in the matrix. Each row of the matrix is allocated a "weight" as shown below. Consider as an example the Greek character "Σ".

64	X	X	X	X	X	X
32		X				
16			X			
8				X		
4			X			
2		X				
1	X	X	X	X	X	X
	65	99	85	73	65	65

Looking at the left-hand column, we see that the squares corresponding to 1 and 64 are occupied. The number representing this column is therefore 1 + 64 = 65. In the second column, squares 1, 2, 32 and 64 are occupied so that the number for this column is 1 + 2 + 32 + 64 = 99. The remaining four numbers are calculated in the same way.

The character can then be sent to the printer using secondary address 5

```
OPEN5,4,5
C$=CHR$(65)+CHR$(99)+CHR$(85)+CHR$(73)+CHR$(65)+CHR$65)
PRINT#5,C$
CLOSE5
```

Once this has been done, the character may then be printed; it is represented by CHR$(254).

```
OPEN4,4
PRINT#4, CHR$(254)
CLOSE4
```

The method shown above is very cumbersome, and makes it difficult to alter the stored special character; it is more convenient to input the character by means of a DATA statement. For example:

```
 100  DATA65,99,85,73,65,65
 200  OPEN5,4,5
 300  FOR I=1 TO 6
 400  READ C
 500  C$=C$+CHR$(C)
 600  NEXT I
 700  PRINT#5,C$
 800  OPEN4,4
 900  PRINT#4,CHR$(254)"(X*Y+Z)"
1000  CLOSE4
1100  CLOSE5
1200  END
```

A number of different special characters may be used, each being selected simply by changing the data in the DATA statement. Try your hand at generating other symbols – for example a £ symbol, or a face.

Some difficulties may arise if attempts are made to print a special character continuously, but where the special character is used only as part of printed text, no problems should be encountered. As usual with the PET printer, this is due to the bug which crops up whenever two PRINT# statements to logical files with secondary addresses are carried out in succession. As usual also, this may be avoided by interpolating a PRINT# statement to a logical file *without* a secondary address between the two offending instructions.

8.8.6 Line separation – secondary address 6 (tractor feed models only)

In the tractor-feed printers, the paper is advanced in a series of small steps – 144 steps/inch for the 2022 and 3022 printers, and 192 steps/inch for the 4022.

The number of steps between successive lines of printing can be set by means of a logical file with a secondary address of 6.

Where the line separation is not specifically set in this way, the printer will print at a default value of 6 lines/inch – i.e., 24 steps/line on a 2022 or 3022, and 36 steps/line on a 4022.

Once set, the spacing remains at the new value until either it is reset or until the printer is switched off.

It is therefore good practice to reset the spacing to the standard value at the end of your program. Where the printer is often used with non-standard line spacings, it may in any case be worthwhile to reset the spacing to the standard value at the beginning of every program, just in case the spacing is already set to some other value. The sequence of commands required to set the line spacing is:

```
OPEN6,4,6
PRINT#6,CHR$(48)
CLOSE6
```

The value of 48 used here will reset the line spacing to 48 steps/line – i.e., 3 lines/inch on a 2022 or 3022, and 4 lines/inch on a 4022 printer. Clearly other values may be used at will. The sequence of commands may be input either as part of a program or direct from the keyboard.

Note however that as in the case of the formatting facility, difficulties arise where successive PRINT# statements with secondary addresses follow immediately upon one another. To avoid this, some other PRINT# statement must be directed to the printer between instructions to alter the line spacing. This precaution is particularly important where the spacing is reset at the beginning and end of the program, since otherwise problems may be encountered if the program is run repeatedly.

Where programs involve a large number of PET graphics symbols, it may be useful to print the listing at 8 lines/inch, rather than the standard 6. This eliminates the gaps between the symbols on successive lines. The adjustment to the line spacing is readily carried out from the keyboard before printing the listing.

Note that some difficulties may arise if the paging facility is used with a non-standard line-spacing; the effective page length may need to be reset if paging is to operate correctly.

8.9 ADDITIONAL FEATURES OF THE 4022 PRINTER

The facilities discussed so far are available on the 2000-, 3000- and 4000-series printers alike. However, the 4022 printer also has some additional facilities, invoked by using the secondary addresses 7-10, which are not available on the earlier machines.

8.9.1 Lower Case/Upper Case – secondary address 7

The printer normally prints in upper case (capital) letters: where a shifted character is encountered, the corresponding graphics symbol is printed, just as on the PET screen.

However, we already know that the character set of the screen display can be changed, by means of the instruction POKE 59468,14. In 2000-, 3000-, and 4000-series PET's, this instruction causes unshifted characters to be printed on the screen in lower case and shifted characters in upper case, just as with a conventional typewriter.

The 4022 printer can similarly be switched to an alternative character set, by using a secondary address of 7. For example, the instruction sequence

```
OPEN7,4,7:PRINT#7:CLOSE7
```

will cause the printer to reproduce unshifted characters in lower case, and shifted characters as upper-case letters and numerals, rather than the corresponding graphics symbols. It is clearly desirable that the printer and PET should operate in the same manner, so that the complete sequence of instructions would normally be:

```
POKE 59468,14:OPEN7,4,7:PRINT#7:CLOSE7
```

8.9.2 Upper case/graphics – secondary address 8

The PET can be returned to its normal mode, printing only upper-case characters and graphics symbols, by means of the instruction POKE 59468,12. (2000-, 3000- and 4000-series machines).

Similarly the printer can be reset to the upper case/graphics mode by writing to a secondary address of 8. Again, it is desirable that both PET and printer should be operating in the same mode, so that an appropriate instruction sequence would be:

```
POKE 59468,12:OPEN8,4,8:PRINT#8:CLOSE8
```

8.9.3 Disable error Messages – secondary address 9

If the printer error message facility has been enabled, it may be disabled by writing to a secondary address of 9. For instance:

```
OPEN9,4,9:PRINT#9:CLOSE9
```

8.9.4 Resetting the printer – secondary address 10

If we write to a secondary address of 10, all the special functions are reset to their original state, just as if the printer were switched off and on again. This is a particularly useful facility, since such quantities as the line spacing may be conveniently reset to their standard values by a single line at the end of a program.

For instance:

```
OPEN10,4,10:PRINT#10:CLOSE10
```

Note: In many cases, it is necessary to run a program on a variety of systems. Where this is so it may be better to leave these additional facilities of the 4022 printer unused, so that your program will be "portable" between systems using any of the 2000-, 3000- or 4000-series printers.

CHAPTER 9

Disk Units

9 Disk Units

9.1 FLOPPY DISKS

The "floppy disk" now so widely used for bulk storage is a disk of thin flexible plastic coated with a magnetic material similar to that used on magnetic recording tape. Two standard sizes of disk are available – 8-inch and 5¼-inch. The smaller size, the "mini-floppy" or "diskette" is almost universally used in microcomputer applications. The disk is sealed inside a stiff protective envelope, with openings through which the mechanical drive unit and the magnetic recording head of the disk drive can make contact with the disk. The disk never leaves the envelope; the envelope remains stationary while the disk revolves within it to allow the head to access points over the whole of the working surface. At one side of the envelope is the "write protect" notch. If this is covered the disk contents cannot be altered.

Data and/or programs can thus be stored magnetically on the disk in precisely the same manner as on cassette tape. However, the capacity of the disk is greater – roughly that of a full C90 cassette – and the speed of recording and reading is between 50 and 100 times greater. The reliability of reading and recording is also much better than for cassettes.

Several types of disk drive units have been produced by CBM, and a twin drive unit produced by Computhink is also in fairly common use. The Computhink system has been largely superseded by the latest CBM units which have superior disk handling facilities. The earliest CBM units, the 2040 and 3040 were twin drive systems with a Disk Operating System called DOS 1. These systems use rather cumbersome operating techniques, but the more recent 4000 and 8000 series computers have disk control commands incorporated into BASIC 4. The 4040 disk drives operating with DOS 2 are identical in many respects to the 3040 drives and it is possible to upgrade the 3040 by replacing the ROM chips so that BASIC 4 commands can be employed.

CBM have recently introduced a more economical single drive unit (the 2031 system) which operates with BASIC 4 and DOS 2.

The use of disk drives, particularly the early units, is greatly simplified by employing a utility program supplied by the manufacturers. According to the version, this program is called "DOS SUPPORT" or "UNIVERSAL WEDGE". The operation of the program is described later in this section and it is well worth the time spent in learning the somewhat arbitrary syntax.

The use of all these disk drive units is described in the following sections.

9.2 THE DISK DIRECTORY

Any disk drive unit contains a complex microprocessor-based control system which operates according to a built-in program – the disk operating system (DOS).

One of the functions of the DOS is to keep track of the identity of the disk and of the files (programs or data) which it carries. To this end, a few data blocks (roughly a thousand bytes – equivalent to about two hundred words in English) are dedicated to storing the identifying name and

number of the disk, together with a list of the names and sizes of the files stored upon it, and the amount of space remaining. This information constitutes the *disk directory*, and may be examined by the user via the microcomputer screen.

9.3 USING 2000 AND 3000 SERIES CBM DISK DRIVES

a) Check that both the PET and the disk drive unit are turned OFF.

b) Make sure that the disk drive unit is connected to the IEEE port of the PET – i.e. the connector nearest the centre of the machine, at the back.

c) Switch on the PET.

d) Open the disk drive doors and CHECK THAT THERE ARE NO DISKS IN THE DRIVE. The CBM drive unit must *NEVER* be switched on with a disk or disks in position. If this is done it is almost certain that any program or data stored on the disk will be corrupted.

e) Switch on the drive, using the mains switch in the back panel of the disk drive unit. The LED's in the front panel should glow briefly and then go out.

f) Insert the disk or disks and close the drive door(s). The disk must be inserted with the label on top and nearest the user.

9.3.1 Loading a program from disk into PET memory

Assuming that a disk is available on which programs have previously been saved, the following procedure should be followed. (The explanatory notes in each sub-section need not be read the first time round – the instructions may be regarded simply as a recipe.)

a) Insert the disk in the drive.

b) Type in the command

OPEN15, 8, 15

The first of the numbers in this command need not be 15 – any number up to 255 can be used; however the same number must be used in the PRINT # commands used later. The other two numbers, 8 and 15, must not be changed. (See Chapter 7.)

This command opens a channel of communication between the PET and the disk drive. If you subsequently input an incorrect command, the channel ("file") will automatically be closed, and must be re-opened, using the above command, before proceeding further.

c) "Initialize" the disk by typing

PRINT#15,"I0"

or PRINT#15,"I1"

according to which drive is being used. If both are to be initialized, this may be done by means of the command

PRINT#15, "I"

The INITIALIZE command aligns the drive unit with the tracks on the disks, reads the directory from the disk and loads the disk identification data into the DOS. This command must be used whenever a fresh disk is placed in the drive, or when a disk is removed and replaced.

[For best results, open the drive doors before giving the INITIALIZE command, and close each

door a couple of seconds after the appropriate indicator LED lights up, showing that the drive is in use. This procedure helps centralise the disk.]

d) The selected program may now be loaded using the command

LOAD"drno:progname",8

drno is the number (0 or 1) of the drive in use, and progname is the name of the program to be loaded – e.g.

LOAD"0:MATMULT",8

The program is now loaded into the microcomputer, any BASIC program already stored there being automatically erased. When loading is complete (after a few seconds) the program may be RUN, LISTed or modified in the usual way. Other programs may be LOADed from the same disk without re-initialization.

9.3.2 The disk directory

The disk directory may be inspected by means of the command

LOAD"$drno",8

When loading is complete the directory may be displayed on the screen by means of the command LIST.

9.3.3 Saving a program on disk

Where the disk has previously been used but is not full, this is quite straightforward.

a) The disk is inserted in the drive and initialized as before, using the appropriate command

PRINT#15,"I"

PRINT#15,"I0"

OR

PRINT#15,"I1"

b) The program is then SAVEd using the command

SAVE"drno:progname",8

9.3.4 Verifying a saved program

After being SAVEd, the program may be verified. This process checks the SAVEd program against the original stored in the PET, to check that it has been correctly written to the disk.

The appropriate command is:

VERIFY"drno:progname",8

Note that the commands LOAD, SAVE and VERIFY all have exactly the same format.

Where a program is verified immediately after being saved, the abbreviated command

VERIFY"*",8

may be used.

It is good practice to VERIFY every program as it is SAVEd.

9.3.5 Using a new disk (or erasing and re-using an old disk)

Where a disk has not previously been used or is to be completely erased and re-used, the procedure is a little lengthier. Before being used, the disk must be "formatted"; this process divides the disk into sectors, imprinting magnetic signals upon it to indicate the limits of each sector.

To do this the disk is placed in the drive. The command OPEN 15,8,15 is given, if this file has not already been opened, and the disk is formatted by means of the command

PRINT#15,"Ndrno:diskname,id"

diskname is any name up to 16 characters long, and id is a two-character identifier; both may be selected at will by the user. ('N' stands for NEW).

9.3.6 Other Commands

It is good practice to close the link between computer and disk drive after use by means of the command

CLOSElfn

(lfn is the "file number" used in the original OPEN statement.)

A number of other commands using the format

PRINT#lfn," . . . . . "

are available: we have already seen the use of the INITIALIZE and NEW commands. Other commands which may be used when the user has developed some confidence are:

D – DUPLICATE: duplicates an existing disk

C – COPY: copies files, either on the same or different disk

R – RENAME: allows the name of an existing file to be changed

S – SCRATCH: deletes a file

V – VALIDATE: creates a map of the available blocks on the disk, and initializes the disk.

FOR FURTHER DETAILS ON THE USE OF THESE COMMANDS, THE CBM FLOPPY DISK USER MANUAL SHOULD BE CONSULTED.

9.4 USING THE 4000 SERIES CBM DISK DRIVE

The 4000 series of CBM computers and accessories using BASIC 4.0 (introduced late 1980) are provided with an additional set of DOS commands. These are incorporated in the computer ROM and can be regarded as an extension to the BASIC peripheral commands.

Compatibility

All the DOS 1 commands described in the previous section can be used with the 4000 series drive. A program which has been saved using a 3000 series disk drive can be loaded by means of a 4000 drive unit; however, 3000 series formatted disks cannot be used for *saving* additional programs with a 4000 series unit.

9.4.1 Starting up

The starting up procedure is identical for both disk drive series. It is important never to switch the disk drive on with a disk in position.

9.4.2 Loading a program from disk into PET memory (4000 series)

If a unique ID is assigned to each disk no initialization procedure is required with the 4000 series unit.

The command

DLOAD"progname",Ddrno

will cause the program entitled "progname" to be loaded into PET memory.

The drive number (drno) will default to 0 if not specified:

DLOAD"progname"

will load the desired program from drive number 0.

The 4000 series DOS also contains a useful quick load procedure:

Switch the computer and the disk drive unit on and insert a disk into drive 0. Simultaneously press the SHIFT and RUN/STOP keys. The computer will initialize the disk in drive 0 and load the first program into PET memory.

9.4.3 The disk directory (4000 series)

The disk directory may be inspected by means of the command

DIRECTORYD0 or DIRECTORYD1.

This format can be shortened to:

DI_D0 or DI_D1

A shifted R produces the "_" in the short format. Typing

DI_

will display the directory for disks in both drives. The command CATALOG (or C shifted A) can alternatively be used.

During the listing of the directory, the space key can be used to stop and then restart the listing. This is particularly useful because a large directory, when listed, will scroll off the screen.

9.4.4 Saving a program on disk (4000 series)

Where a disk has previously been used but is not full, the procedure is very straightforward. Normally, no initialization is required.

The command

DSAVE"progname",Ddrno

will save the file (progname) on drno 1 or 0. The program name may be up to 16 characters long.

9.4.5 Verifying a saved program (4000 series)

The format of the VERIFY command for the 4000 series is the same as that for the 3000 series.

VERIFY"drno:progname",8

or

VERIFY"*",8

9.4.6 Using a new disk (4000 series)

Before being used for the first time, the disk must be "formatted".

This is done with the aid of the HEADER command. The format is:–

HEADER "dn",Ddrno,Iid

where dn = disk name supplied by the user and limited to 16 characters.

drno = drive number, 1 or 0
id = the disk ID, which is a unique two character alphanumeric identifier, supplied by the user.

9.4.7 BACKUP (4000 series)

This command can be used to create a backup copy of a disk and has the format

BACKUP Dsdr TO Dddr
sdr – source drive (either 0 or 1)
ddr – destination drive (either 0 or 1)

BACKUP can only be used with disks formatted on a 4000 series machine.

9.4.8 COPY (4000 series)

With a formatted disk (using the HEADER command) COPY can be used to copy either the whole of the contents or a single program from one disk to another.

COPY Dsdr TO Dddr

can be used to copy the whole disk.

COPY Dsdr, "progname" TO Dddr,"progname"

will copy a specified program.

again sdr is the source drive, either 1 or 0.
again ddr is the destination drive, either 1 or 0.

COPY can be used to copy 3040 formatted disks onto a 4040 formatted disk.

9.4.9 SCRATCH (4000 series)

This command is used to erase a specified program. The format is:–

SCRATCH Ddrno,"progname"

where drno = 1 or 0.

Not all of the available BASIC 4.0. commands have been described in this brief introduction to the use of the 4000 series disk drive unit.

Other commands are available for file and data management and these are referred to in the chapter relating to disk file handling procedures.

9.5 DOING IT THE EASY WAY – USE OF THE UNIVERSAL WEDGE PROGRAM

The reader will by now have realised that the use of the commands listed above is tiresome and liable to result in error. The situation may be greatly improved by means of a support program which extends the facilities of the disk operating system. Such a program – UNIVERSAL WEDGE – is

supplied on disk by CBM, and *it is so suitable for the purpose that the operator should consider storing UNIVERSAL WEDGE as the first file on every disk.* (It occupies only 6 of the available 690 data blocks). The program is called DOS SUPPORT on the utility disk which is supplied with earlier machines.

9.5.1 Loading UNIVERSAL WEDGE

If UNIVERSAL WEDGE, in whatever version is available, is stored as the first file on a disk, and the disk is inserted in drive 0 (observing the precautions listed in section 9.1), the process of opening files, etc, may be avoided. Immediately after turning on the drive and inserting the disk simply type in the command

LOAD"*",8

and when loading is complete, type in the command RUN. The quick load procedure (SHIFT RUN) can be used with 4040 drives.

This special LOAD command opens the file, initializes drive 0 and loads the first file (in this case UNIVERSAL WEDGE) from the disk in drive 0. Once UNIVERSAL WEDGE has been LOADed and RUN, the format PRINT#,"..." is no longer required. A command requiring this format will be obeyed if it is preceded by either of the symbols

@ or >

BUT note that other commands such as SAVE and VERIFY must be typed in as before, using the format

SAVE
or VERIFY } "drno:progname",8

9.5.2 Use of UNIVERSAL WEDGE

For example, the command

>I1

will cause drive 1 to be initialized; the command

@I

will cause *both* drives to be initialized.

The commands

>$0 >$1 or >$

will cause respectively the directories of the disks in drive 0, drive 1, or both drives to be displayed.

[An incidental advantage here is that when this command is used to access the directory, any program already in the machine is not over-written and lost, as is the case when the LOAD and LIST commands are used to display the directory.]

9.5.3 Loading a program

Under UNIVERSAL WEDGE, a program may be loaded simply by typing a slash (/) followed by the program name. The drive number may be specified or not, as desired:

/progname or /drno:progname

e.g. /CHEMSIM or /0:SPACE INVADERS

9.5.4 Loading and running a program

When a program has been loaded, it can be RUN in the usual way; alternatively, if an up arrow (↑) is used in place of a slash, the program will be loaded, and then automatically RUN as soon as loading is complete. For example

↑SPACE INVADERS or ↑1:CHEMSIM

9.5.5 Abbreviated program names

When using these two commands (/ and ↑) under DOS SUPPORT or UNIVERSAL WEDGE, the program name need not be spelt out in full. Only the first letter or so need be typed, followed by an asterisk:

/CHEMSIM	or	/CHEM*
↑SYSTEM	or	↑SY*
↑SPACE INVADERS	or	↑S*

Note however that the machine will load the first program it comes to whose name meets this abbreviated specification. Thus the command

/S*

will cause the machine to load the first file it finds whose name begins with S. Enough letters should therefore be given to distinguish the file from any other which may be present on either drive.

To avoid difficulties of this type, a special form of the directory command may be found useful: this is of the form

@$drno:string*

e.g. @$:CH* @$1:SIM* or >$:CHEM*

This command causes to be displayed all file names on the drive specified (or on both drives, if no drive number is given) which begin with the letter or string of letters included in the command.

9.5.6 COPY and SCRATCH with UNIVERSAL WEDGE

The "copy all files" command may be abbreviated to:–

>C1 = 0

or @C1 = 0

Either of these formats will copy all files from drive 0 to drive 1.

Also

S1:AA* is the shortened version of SCRATCH the program (or programs) on drive 1 which start with "AA".

9.5.7 Error messages

When an error message appears on the screen, the use of the command

> or @

will cause an explanatory message to be displayed (although in fact this may be of little help!).

9.6 USING THE COMPUTHINK DISK DRIVE

The Computhink disk drive unit is also designed for use with five inch mini floppy disks. The unit operates with the aid of a ROM based system called 'DISKMON" and users are advised to read the DISKMON USER'S GUIDE which is supplied with the system.

The following set of instructions constitutes a brief summary of DISKMON commands and is designed to help the beginner get off the ground.

9.6.1 Starting up

(a) Check that both the PET and the disk drive unit are switched off.

(b) Switch on the PET.

(c) Switch on the disk drive unit. (When switching off, switch off the disk drive unit first.)

(d) To initialize the "DISKMON" operating system type the instruction SYS 45056 followed by the RETURN key.

(e) Insert the disk or disks and close the drive door(s). The disk should be inserted with the manufacturer's label on the bottom right hand corner nearest the user.

9.6.2 Using a new disk

Before being used for the first time the disk should be "formatted"; this process divides the disk into 40 sections and normally not more than one program (file) is stored in each section. Long programs may take more than one section.

The "format" command is

```
$F,drno
```

drno will be either 1 or 2. The left-hand drive is number 1.

NOTE. The formatting procedures of the Computhink and the CBM drive units are different and this means that disks are not interchangeable.

9.6.3 The disk directory

The directory may be inspected by means of the command

```
$D,drno
```

9.6.4 Loading a program from disk to PET memory

A program may be loaded by using the command

```
$L,drno,"program name"
```

The program name will usually be established by referring to the directory.

9.6.5 Saving a program on disk

A program in PET memory may be saved on disk by using the command

```
$S,drno,"program name"
```

Program names may be up to 16 characters long.

Programs may also be saved in place of existing programs by using the same name. This effectively erases the original.

9.6.6 Erasing a program from disk

A program may be erased by using the command

$E,drno,"program name"

The DISKMON operating system also supports a number of additional commands and BASIC instructions, which may be used when the operator has had some experience. The DISKMON USER'S GUIDE should be consulted for further information.

The Computhink system is supplied with a demonstration disk which contains a number of utility programs. The most useful program for the beginner is perhaps DISKCOPY which enables the full contents of a disk to be copied directly on to a second disk.

CHAPTER 10

Data Storage on Cassette

10 Data Storage on Cassette

10.1 DATA FILES AND THE CASSETTE RECORDER

In order to communicate with peripherals such as the disk drive or cassette recorder, the PET uses logical files. Logical files are fully described in Chapter 7 so in this section we shall merely introduce the syntax and describe the procedures which are used to write and read data files on tape.

Of course, the simplest use of the cassette recorder is for the storage of programs ("program files") with the aid of the familiar LOAD and SAVE commands. In this case, the user is not concerned with the methods which the PET employs in order to organise the program into the individual bits of data which are sequentially stored on the tape. The use of the "data file" is an alternative approach to information storage on tape. The user can save large numbers of data items (numbers and/or strings) in an economical way and can also exercise a greater amount of control over the format of the data on the tape.

The"data file" is usually created and managed with the aid of a BASIC program and there are several commands in PET BASIC which are reserved for use with data files.

10.2 THE OPEN SYNTAX FOR CASSETTE DATA FILES

In this section we shall first describe the syntax which is used to manage a data file and then give examples of typical programs which will enable the user to write data files involving numbers and strings as data items.

It is first necessary to OPEN a logical file through which we will communicate with the cassette recorder. The OPEN command not only opens the file to a specified peripheral, but also determines the mode of operation of the peripheral. The general form of the OPEN command here is:–

OPEN lfn,pa,sa,"filename"

lfn is the logical file number which is chosen by the programmer and may take any value in the range 1 to 255. In practice low numbers are used and the programmer often chooses to make the logical file number the same as the device number.

pa, the primary address or device number is fixed by the manufacturer. For the cassette recorder pa can take the value 1 or 2.

The PET has two cassette ports, with device numbers 1 and 2, into which the user can plug a cassette recorder. The second cassette port is sometimes (on 3000 series models) located inside the computer and the user would require a screwdriver in order to use the facility. This port is located on the right hand side of the machine chassis for 4000 series models. Early PETs with built-in cassette recorders made use of an internal connection port; the built-in cassette is device number 1.

The sa part of the OPEN command is the secondary address. The number assigned here determines the mode of operation of the peripheral. In this case we need to establish whether the cassette tape is to be "written" or "read" and the "sa" value can be 0, 1 or 2. A value 0 corresponds to the "read" function (LOAD for a program file) and the PET is instructed to load the data contained on a data file which has previously been stored on tape. The sa defaults to 0, so that if no value is specified and no filename is supplied the PET will read the first data file encountered on the tape.

A value of 1 corresponds to the WRITE function (SAVE in the case of a program file). If a value of 1 is chosen, the PET will automatically put an end-of-file (EOF) marker on the tape at the end of the data when the file is written. If necessary, further files can then be saved on the same tape.

A secondary address value of 2 also corresponds to the WRITE function but in this case the PET puts an end-of-tape (EOT) marker at the end of the data.

10.2.1 Examples of OPEN commands used for tape management

```
OPEN1,1
```

The logical file number here is 1.
The device number corresponds to cassette port 1.
No secondary address is specified so a default value of 0 is assumed, i.e. the instruction is to read a data file. No filename is specified so the PET will read the first data file encountered on the tape.

```
OPEN1,1,0,"NAME"
```

In this case the full syntax is used and the PET is instructed to read from tape the data file called NAME.

```
OPEN1,1,1
```

Here we specify, by giving the secondary address a value of 1 that we require to write a file with an end of file marker.

```
OPEN1,1,2,"NAME"
```

The secondary address value 2 indicates that we wish to write a data file called NAME, with an end of tape marker.

```
OPEN9,2,1,"NAME"
```

Here the logical file number has been chosen to be 9, the device number corresponds to the second cassette port and the write instruction is given, together with a file name.

After issuing the OPEN command it is necessary to include in the program a separate instruction to write some specified data on to the tape. This instruction is the PRINT#command which must be typed in full and can *not* be abbreviated to ?#. The full format of this command is

```
PRINT#lfn,variable
```

The file number must be the same as that specified in the OPEN command and the variable can be either a number or a string.

Finally it is necessary to CLOSE the logical file, signifying the end of the data file. The PET will incorporate either an end-of-file or an end-of-tape marker on the tape when the file is closed, according to the secondary address used in the OPEN statement.

10.3 WRITING A DATA FILE

The following program indicates how these file management commands can be used to create a simple numeric data file.

Example 10.1

```
100 REM WRITE DATA FILE ON TAPE
110 REM ************************
120 REM      NUMERIC DATA
130 REM      ************
140 :
150 OPEN1,1,1
160 FOR N=1 TO 15
170 PRINT#1,N
180 NEXT N
190 CLOSE1
200 PRINT"DATA FILE COMPLETE"
210 END
```

In this introductory example we have used the OPEN syntax without a file name; the secondary address used ensures that an end of file marker is sent to the tape when the file is closed. The FOR . . . NEXT loop increments N from 1 to 15 and each number is sent to the tape as a separate data item. There is a carriage return between each item; the PET therefore sends a carriage return marker to the tape so that each item can be identified separately when the tape is read.

An interface block of memory called the cassette buffer exists between the cassette recorder and the main PET memory. During the write process the PRINT # command transfers data to the buffer and when the buffer is full a block of data (191 bytes) is sent to the recorder. The tape spools stop periodically whilst the cassette buffer is accepting data. When the file is closed the last items of data are sent to the tape and the recorder motor is switched off.

Data items may be sent as strings and the next example illustrates a simple technique for writing a string data file.

Example 10.2

```
100 REM  WRITING DATA FILES ON TAPE
110 REM  ***************************
120 REM        FOR STRING DATA
130 REM        ***************
140 :
150 OPEN2,1,2,"STRING FILE"
160 FOR S=1 TO 6
170 READ A$
180 PRINT#2,A$
190 NEXT S
200 CLOSE2
210 PRINT"STRING FILE COMPLETE"
220 DATA WRITING,SOME,STRINGS,ON,TAPE,DEMO
230 END
```

In this example the OPEN format includes a file name and the secondary address 2 ensures that an end of tape marker is sent to the tape.

The string items are managed with READ . . DATA commands and as in the previous example the items are separated by carriage returns.

It is also possible to write a data file which contains more than one string entry per record by introducing item separators, as illustrated in the following example.

Example 10.3

```
100 REM WRITE DATA FILE
110 REM ***************
120 REM WITH ITEM SEPARATORS
130 REM ********************
140 :
150 OPEN1,1,1,"STRINGS"
160 FOR J=1 TO 3
170 INPUT A$,B$
180 PRINT#1,A$;",";B$
190 NEXT J
200 CLOSE1
210 PRINT"DATA FILE COMPLETE"
220 END
```

Two strings are entered by means of the INPUT command at line 170. The PRINT# statement contains the two strings separated by a comma (in quotation marks) and semicolons. The comma in quotes could be replaced by CHR$(44) which is the equivalent ASCII code representation.

10.4 READING A DATA FILE

In order to read a data file the OPEN command must contain a secondary address of 0 and the PRINT# command be replaced by an INPUT# command. Otherwise the programming techniques are similar to those employed in the writing process.

Example 10.4

```
100 REM READ DATA FILE ON TAPE
110 REM ***********************
120 REM       NUMERIC DATA
130 REM       ************
140 :
150 OPEN1,1
160 FOR N=1 TO 15
170 INPUT#1,P
180 PRINTP
190 NEXT N
200 CLOSE1
210 PRINT"DATA FILE READ COMPLETE"
220 END
```

This example demonstrates a simple program which can be used to read the numeric data file which was produced in Example 10.1. The numeric data are printed on screen at line 180.

Example 10.5

```
100 REM  READING DATA FILES ON TAPE
110 REM  ***************************
120 REM         STRING DATA
130 REM         ***********
140 :
150 OPEN2,1,0,"STRING FILE"
160 FORI=1TO6
170 INPUT#2,A$(I)
180 PRINTA$(I)
190 NEXT I
200 CLOSE2
210 PRINT"STRING FILE COMPLETE
220 END
```

The string data file which was written in Example 10.2 is read by means of the INPUT#2 command at line 170. The incoming data items are stored in an array and the strings can then be ordered, printed etc., as required.

The next example shows a method of reading the string data file which included item separators.

Example 10.6

```
100 REM  READ  DATA FILE
110 REM  ***************
120 REM WITH ITEM SEPARATORS
130 REM ********************
140 :
150 OPEN1,1,0,"STRINGS"
160 FOR J=1 TO 3
170 INPUT#1,A$,B$
180 PRINTA$;" ";B$
190 NEXT J
200 CLOSE1
210 PRINT"DATA FILE READ COMPLETE"
220 END
```

GET# is an alternative form of INPUT#-GET# transfers one character at a time through the input buffer into memory and unlike INPUT# does not fill up the 191 bytes memory in the input buffer. It is possible to use GET# in order to check for individual items of data in the file; this can provide a useful data management facility.

CHAPTER 11

Data Files on Disk

11 Data Files on Disk

11.1 SEQUENTIAL, RELATIVE AND USER FILES

Data, as well as programs, can be stored on disk. There are three main types of data file available to the PET user – sequential, relative and user files. Of these, the relative file is available only to users of BASIC 4.0 together with Disk Operating System 2.0 or 2.5, i.e. on 4000 and 8000 series machines, or on upgraded earlier models.

In any file, there will be a number of "records". A record is the electronic equivalent of a file card, and consists of a number of items of data. For instance, in an address file, each record might contain the name, address and telephone number of one person. Each separate item within a record is termed a data "field" – thus the name, address and telephone number in our address file would each occupy a separate field. Clearly the lengths of the different fields in a record will not usually be the same.

In a sequential file it is not possible to access a single item from the file directly; the file must be read through from the beginning to identify the appropriate entry. If the file is sufficiently small to be stored in its entirety in machine memory this may not present a major problem, but for very large files it may be a serious disadvantage. In many applications, however, "random access' or "direct access" to individual records is not required, and in such a case a sequential file may be perfectly satisfactory. Examples might be a school class list, or the payroll of a firm, where it is in any case necessary to access every item on the file each time the file is used. Sequential files have the by no means negligible advantages that they occupy significantly less space than do direct-access files and that when one item has been dealt with, no time is wasted searching for the next entry. Historically, the widespread use of sequential files is at least partly due to the use of such data storage media as magnetic tape or punched paper tape, where access to data is by the very nature of the medium sequential.

Once the length of the file becomes greater than about 30 kbytes, it will usually be found worthwhile to use a random-access file rather than a sequential file, except for purely sequential applications such as those mentioned above. There are two types of random-access or direct-access file – "relative" and "user" files.

In a relative file the user can pick out any record from the file relative to the beginning of the file. This gives the user direct access to any desired record; he can also, if desired, read through the records sequentially, just as with a sequential file. The major disadvantage of the relative file in comparison with the sequential file is that it will normally occupy significantly more space on the storage medium. This is because it is usual to allow the same amount of space for every record, regardless of the actual number of characters to be entered. Thus, in our address file, the space allocated to the name must be sufficiently large to hold the longest name likely to be encountered. (If you have friends or customers with names like "Montmorency Marmaduke Featherstonehaugh-Urquhart", this can clearly be a serious problem, since John and Fred Smith will have to be allocated just as much space on the disk as the polysyllabic gentleman aforesaid.) To make the records of variable length would be very complicated, especially when entries are liable to be changed at a later date. (In a sequential file, where each record is allocated precisely the required amount of space, the whole file must be rewritten whenever an entry is to be updated.)

For those unfortunates without BASIC 4.0, it is possible to construct direct-access files, but the process is rather complicated, since the program must allocate in detail the disk space required for

each entry, and keep a detailed record of the position on the disk of each item of information. The creation of such "user" files is dealt with later, but if considerable work is to be carried out with random-access files, it will almost certainly pay the user to have his system upgraded. A further disadvantage of user files is that, unless the user is prepared to indulge in some rather complicated programming, they tend to occupy considerably more space even than relative files.

11.2 INDEXING RANDOM-ACCESS FILES

When accessing direct-access files, we may wish to select records according to a number of possible criteria. In the case of a small file, we can just read the appropriate field of every record in turn – rather like searching a sequential file. The difference is that with a sequential file the whole of each record must be read in turn, whereas with a direct-access file we need access only the appropriate field of each record. Where the file is large, however, this can take an excessively long time.

Where a large file must be repeatedly searched in this way, it is convenient to *order it.* However, each record may contain a number of fields, and on different occasions we may need to search the field for different characteristics. For example, in a library catalogue we may search for a book by name, or by subject, or by the author's name. Similarly, if a general practitioner were to keep a record of his patients, he might need to search the record by name or age, or by the patients' ailments, or by the drugs prescribed.

In such a case it is usual to prepare a separate index for each characteristic or 'key'; each index is then sorted into alphabetic or numerical order to simplify future reference. Of course, each index is itself a separate file, which must be separately stored. These index files will in general be much shorter than the main file however, since each entry will consist only of the key and the file record number. According to the size of the index file, it may be either sequential or relative. A sequential file will in general occupy less space, but must be searched in strict sequence, whereas more sophisticated search strategies – e.g. the binary search – may be used on a relative file. In a binary search, the file is repeatedly divided into two, and a decision is taken as to which half contains the first of the wanted entries. This process is repeated until the first of the desired entries is located. The process is essentially similar to that unconsciously adopted by most people when seeking a word in a dictionary. The equivalent of a sequential search would be to scan each word in turn, starting with 'aardvark' and continuing until the required word is found – clearly not the speediest method, although very reliable, and easier to program.

11.3 ERROR MESSAGES

Should something go wrong with an attempt to carry out any disk operation, the disk controller can send an error message to the PET.

With BASIC 4.0 the process is quite straightforward, since the error message consists of a single string which can be read directly, without opening a special command and error channel to the disk controller. This is done by means of the command

```
PRINTDS$
```

Since each of the various possible error messages begins with a different number, it is often convenient to read only the numerical part of the message, using the numerical variable DS; messages with numbers below 20 can in general be ignored. The skeleton of a typical BASIC 4.0 disk error subroutine would thus be:

```
nnn     GOSUB 10000
              :
              :
              :
              :
```

```
10000    IFDS<20 THEN RETURN
10010    PRINTDS$
10020    END
```

On systems using BASIC 2.0 the process is somewhat more complicated, since the message consists of three numerical variables and a string variable, and these must be accessed via the special command and error channel (secondary address = 15).

The complete error message consists as before of a number and the error message itself, but two more numbers indicating where applicable the track and sector at which the fault occurred, are also output.

A complete BASIC 2.0 disk error subroutine might be:

```
nnn      OPEN15,8,15
              |
              |
nnn      GOSUB 1000
              |
              |
              |
1010     INPUT#15 EN,EM$,ET,ES
1020     IF EN=0 THEN RETURN
1030     PRINT EN;EM$,ET,ES
1040     CLOSE15
1050     END
```

In many applications a knowledge of the track and sector where failure occurred, and of the error message number, is not helpful. In this case line 1030 may be simplified to

```
1030     PRINT EM$
```

11.4 USING SEQUENTIAL DISK FILES

11.4.1 Creating, updating or reading from a sequential file

A sequential file is opened by a command of the general form:

OPEN lfn, pa, sa, "driveno:filename, filetype, mode"

lfn is the logical file number, which as always may be selected by the programmer. It must be in the range 1 - 255.

pa is the device number or primary address – normally 8 for the CBM disk drive.

sa is the secondary address, which may be selected at will by the programmer, within the range 2 - 14.

driveno will be either 0 or 1, according to which of the two drives contains the disk to be used.

The file name must not exceed 16 characters in length.

Since we are here working with sequential files the file type will be SEQ, for sequential (this may be abbreviated to S).

The mode indicates whether the program is to WRITE data into the file or READ from it (usually abbreviated to W or R respectively).

The command for the opening of a new sequential file to be called "NAMES", on a disk placed in drive 0, would thus be

```
OPEN2,8,2,"0:NAMES,S,W"
```

If we later wished to open the same file for reading, we might use:

```
OPEN3,8,3,"0:NAMES,S,R"
```

Of course, the same logical file number cannot be used for two different logical files at the same time, and if a file is to be kept open for both reading and writing, two separate logical files must be opened – one for each mode.

11.4.2 Interactive Commands

The command strings required to open a file may be assembled by *concatenating* shorter strings within the program. A program can thus ask the user for the name of a file, the drive in which the relevant disk has been placed and so on, before OPENing a logical file. This is seen in the demonstration programs WRITEDATAFILE and ORDERPRINT (Examples 11.1 and 11.2).

Once a file has been opened, data are written to or read from it using the PRINT#, INPUT# or GET# commands.

For example:

```
OPEN3,8,3,"0:FILE,S,R"
INPUT#3,Z$(1)
```

or:

```
OPEN6,8,6,"1:ANOTHERFILE,S,W"
PRINT#6,N,Z$,P
```

Example 11.1

WRITEDATAFILE

This demonstration program illustrates the process of creating or updating a file. For simplicity the file used consists simply of a list of strings. This list might for example be a file of customer names, or a school class list.

```
100 REM   ***************
110 REM   *WRITEDATAFILE*
120 REM   *  E.A.FLINN  *
130 REM   *  MAY  1982  *
140 REM   ***************
150
160 REM      THIS PROGRAM CREATES OR UPDATES A SEQUENTIAL DATA FILE
170
180 REM      THE DATA STORED CONSISTS OF A SERIES OF TEXT STRINGS (E.G.NAMES)
190
200 PRINT"■": PRINT:PRINT:PRINT:
210 INPUT "WHICH DRIVE - 0 OR 1";D$
220
230 REM      OPEN ERROR CHANNEL
240 OPEN15,8,15
250
260 REM      INITIALISE DRIVE
270 PRINT#15,"I"+D$
280
290 INPUT "■■■NAME OF FILE";N$
300 X$=D$+":"+N$+",S,W"
310 PRINT:PRINT:PRINT:
```

```
320 INPUT"NEW FILE (N) OR OLD (O)";F$
330 L$=LEFT$(F$,1)
340 IF L$<>"N" AND L$<>"O" THEN PRINT "■■■":GOTO 320
350 REM
360 REM       OPEN FILE
370 REM
380 IF L$="O"THEN PRINT#15,"S"+D$+":"+N$
390 OPEN2,8,2,X$
400 GOSUB 590
410 PRINT"■":PRINT:PRINT:
420 INPUT"NUMBER OF ENTRIES";N
430 PRINT#2,N
440 GOSUB 590
450 PRINT"■":PRINT:PRINT:PRINT:
460 PRINT"■INPUT LIST OF NAMES,TERMINATING EACH BY A CARRIAGE RETURN.";
470 PRINT"■NAMES MUST NOT INCLUDE COLONS OR COMMAS."
480 CR$=CHR$(13)
490 FORI=1 TO N
500 INPUT A$
510 PRINT#2,A$;CR$;
520 GOSUB 590
530 NEXT I
540 CLOSE2
550 CLOSE15
560 END
570 REM       CHECK FOR DISK ERRORS
580 REM       ***** *** **** ******
590 INPUT#15,EN$,EM$,ET$,ES$
600 IF EN$="00" THEN RETURN
610 PRINT:PRINT:PRINT
620 PRINTEM$,EN$ ET$,ES$
630 CLOSE2
640 CLOSE15
650 END
```

The program first asks the user for the number of the drive containing the disk on which the file is to be written (line 210). It next opens a command and error channel to the appropriate disk drive (line 240) and initialises the disk (line 270).

It then asks the user for the name to be allocated to the file, and assembles the string to be used in the OPEN command (lines 290 and 300). If the file is an up-dated version of a file already on the disk, the original version must be deleted or "scratched" (line 380).

The new file is then opened (line 390), using the string assembled from the information which has been input by the user.

Note that at this stage the error channel is checked (line 400). This is done *every time* a disk operation is carried out – i.e. every time an item of information is read from or written to the disk. If a disk error occurs, the logical files which have been opened are closed, and the program ENDS (lines 630, 640, 650).

Now that the file has been opened, we can write data into it. The first item to be stored is the number of entries to be made in the file (lines 420 and 430). This information is not strictly necessary for the process of writing a file, but is useful when the file is to be read back. (This is seen in the next program ORDERPRINT. In ORDERPRINT the entries are read from disk and stored as indexed string variables within the PET; it is therefore necessary to know the number of entries so that the array can be appropriately dimensioned.)

Each entry in turn is then input by the user and transferred to disk (lines 480 - 520), the error channel being checked at every step (line 520) to ensure that no write errors have occurred. Finally the logical files are closed and the program ENDs (lines 540-560).

Note that in line 510, each string written to the disk is followed by a carriage return – CHR$(13).

This is required by the disk drive software, to indicate the end of the entry. The semicolons suppress the normal line-feed with carriage return which would otherwise be written to the disk by the PET at the end of the PRINT# statement. This precaution is not always essential, but is good practice, since otherwise the line feed – CHR$(10) – will be written at the head of the next entry. Often this will not matter, but if the data are to be sorted alphabetically after being read in from disk, and combined with other inputs from the keyboard, the extra character may well upset the sort.

If more than one item is to be entered at a time – e.g. a name and a telephone number – then the separator between the items must be enclosed in quotes: for example

```
PRINT#8,A$","B$;CHR$(13);
```

Where a number is to be written to disk as part of such a command, the number is best expressed as a string, using if necessary the STR$ function. For example:

```
INPUTA$,B
PRINT#1,A$","STR$(B);CHR$(13);
```

N.B. Since this program and the following program ORDERPRINT are intended purely as demonstrations of the techniques necessary to write data to disk and recover it, they have been kept as simple as possible. In a real program, precautions such as those detailed in Chapter 12, "Interactive Programming", would be required. It would also normally be necessary to make provision for NEWing a disk, so as to allow the use of a new and unformatted disk, or the re-use of an old disk.

Example 11.2

ORDERPRINT

This program shows how data stored in a sequential file may be extracted and used. Data are read in from a file created by the use of program 11.1. It is assumed that the strings read in consist of a school class list. The user inputs an examination mark for each name; the students are then sorted into mark order, and their names are printed out, together with their marks and ranking. This shows how even a very simple data file can be utilised in practice. The program also demonstrates the usefulness of the formatting facilities on the PET printer.

```
100 REM  ************
110 REM  *ORDERPRINT*
120 REM  ************
130 REM
140 REM
150 REM  THIS PROGRAM READS IN NAMES FROM AN EXISTING FILE
160 REM
170 REM
180 REM  THE USER INPUTS MARKS, AND THE NAMES ARE PRINTED OUT IN MARK ORDER
190 REM
200 GOSUB 700
210 OPEN15,8,15
220 PRINT#15,"I"+D$
230 X$=D$+":"+N$+",S,R"
240 OPEN2,8,2,X$
250 INPUT#2,N
260 GOSUB 790
270 PRINT"■■NUMBER IN SAMPLE = ";N
280 PRINT
290 PRINT
300 DIM A(N):DIM B(N):DIM Z$(N)
310 FOR K=1 TO N
320 INPUT#2,Z$(K)
```

```
330 GOSUB790
340 PRINT Z$(K);TAB(35) :INPUT A(K):B(K)=K
350 PRINT
360 NEXT K
370 CLOSE2
380 CLOSE15
390 REM       SORT NAMES INTO MARK ORDER
400 REM       **** ***** **** **** *****
410 FOR P=1 TO N
420 FOR K=1 TO N-P
430 IF A(K)>=A(K+1) THEN GOTO 460
440 X=A(K):A(K)=A(K+1):A(K+1)=X
450 Y=B(K):B(K)=B(K+1):B(K+1)=Y
460 NEXT K
470 NEXT P
480 REM       CHECK FOR EQUAL MARKS
490 REM       ***** *** ***** *****
500 FOR K=1 TO N
510 C(K)=K
520 IF A(K)=A(K-1) THEN C(K)=C(K-1)
530 NEXT K
540 REM
550 OPEN 4,4
560 OPEN1,4,1
570 OPEN3,4,2
580 F$="   AAAAAAAAAAAAAAAAAA    999           999"
590 PRINT#3,F$
600 PRINT#4,"   NAME                    MARK          POSITION"
610 PRINT#4,"   ****                    ****          ********"
620 FOR K=1 TO N
630 PRINT#1,Z$(B(K));CHR$(29),A(K),C(K)
640 NEXT K
650 PRINT#4
660 CLOSE 1
670 CLOSE3
680 CLOSE4
690 END
700 PRINT"■   LISTING AND ORDERING PROGRAMME"
710 PRINT"    ******* *** ******** *********"
720 INPUT "■■■NAME OF DATA FILE";N$
730 INPUT "■■■WHICH DRIVE HOLDS DATA FILE - 0 OR 1";D$
740 PRINT"■■■ENTER APPROPRIATE MARK AGAINST NAME■■"
750 PRINT"PROGRAMME ORDERS NAMES IN MARK ORDER,   HIGHEST FIRST"
760 RETURN
770 REM       READ ERROR CHANNEL
780 REM       **** ***** *******
790 INPUT#15,EN$,EM$,ES$,ET$
800 IF EN$="00" THEN RETURN
810 PRINT"■■DISK ERROR■"
820 PRINT EM$,EN$,ES$,ET$
830 CLOSE2
840 CLOSE15
850 END
```

As before, the user is asked to specify the number of the drive containing the disk, and the name of the file to be accessed (lines 720 and 730). These items of information are used (lines 220 – 230) to initialise the drive and to form a string for use in the command which opens the file for reading. The number of names in the file is read in (line 250), and used to dimension the arrays used within the program (line 300). The names are then read in turn, the error channel being checked after every operation, and assigned to the subscripted string variable Z$. Each name in turn is then displayed on the screen so that the user can input the appropriate mark (line 340). When all the marks have been

input the names are sorted into mark order (lines 410-470). Since it is assumed that the size of the class will be reasonably small, speed of sorting is relatively unimportant, and a bubble sort has therefore been used for simplicity. In other applications it might well be necessary to use a speedier and more efficient sort routine.

Once the class has been sorted into mark order, channels to the printer are opened, and the names, marks and class placings are printed out using a suitable format (lines 550-690).

It will thus be seen that sequential data files are relatively straightforward in use, and lend themselves well to many routine clerical or administrative applications.

11.5 RANDOM-ACCESS FILES: RELATIVE AND USER FILES

In a sequential file, such as those dealt with in the preceding section, all the items are read *in order*. Where we wish to inspect or alter only a single record somewhere in the middle of the file, this clearly wastes a considerable amount of time. Once the file exceeds 30-40 kbytes, the delay becomes excessive, and a "random-access" or "direct-access" file will normally be preferred. Even where a random-access file is constructed however – as for a catalogue or a directory – it is quite common for the index file or files, from which the location of the desired record can be found, to be sequential.

Random-access files can be constructed on a PET system – relatively easily on systems using BASIC 4.0, such as the 4000- and 8000-series or updated earlier machines, and with rather more difficulty on unmodified early machines using BASIC 2.0.

On machines using DOS (disk operating system) 2.0 or 2.5, "relative files" can be constructed, using a set of commands not unlike those used in connection with sequential files. On earlier machines it is necessary for the program to take direct control of the process by which data is written to individual locations on the disk, and to keep a record of precisely where on the disk each item of data is stored. The use of such "user files" is clearly a more complicated business, and where the system allows, relative files will normally be preferred for their simplicity. Both user files and relative files are available with DOS 2.0 and 2.5 (i.e. on 4000- and 8000-series machines,) while earlier versions of DOS allow only user files.

11.6 RELATIVE FILES

The principle of a relative file is identical with that of the user files discussed in the next section. However, most of the "housekeeping" required to write data to the disk, and to keep track of the physical location of specific entries, is carried out by the operating system, so saving the programmer a considerable amount of effort.

To simplify the handling of random-access files, whether relative or user, it is usual to allocate the same space on the disk to each record, regardless of possible variations in length of the individual data items within each record. Thus, in an address file, the space allocated to the name will in every case be sufficiently large to hold the longest name likely to be encountered, even though this means that in most cases a considerable amount of disk space will be wasted.

In a sequential file, each entry is given the exact amount of space necessary to accommodate the entry: *a random-access file therefore always occupies more disk space than a sequential file containing the same data*. On the other hand, a single entry in a random-access file can readily be modified, whereas a sequential file must be completely re-written in order to carry out even the most minute alteration. These facts will be seen more clearly after studying the simple demonstration programs which follow. Let us look at the instructions necessary to open, modify and use a relative file.

To create a relative file, we use the command DOPEN#. The file is then "initialised" – i.e. the character CHR$(255) is written into the first character position of the space allocated to each record. (255 is the ASCII representation of the symbol π, which is unlikely to be used in the normal course of

events.) The number of locations reserved at this stage should be adequate to allow for any additions to the file which may later become necessary. This ensures that the disk operating system will reserve sufficient space on the disk for such later entries. Even where space has not been specifically reserved in this way, a file can readily be expanded at a later date. Clearly, however, this process will depend on the availability of sufficient space on the disk, and it is wiser to make adequate provision for expansion when the file is first created.

Once a relative file has been created, records can be entered. This can be done one by one, as required – all the records need not be entered at once.

Once a relative file has been created, it can be re-opened for subsequent access using the normal OPEN command; the command string necessary for this can, as in programs 11.1 and 11.2, be built up from data items entered from the keyboard. Entries in the file may then be made, altered, or simply read, as required.

11.6.1 Creating and Initialising a Relative File

DOPEN# A relative file is created by means of the command DOPEN#, which is available only in BASIC 4.0 (i.e. on 4000- and 8000-series machines, or upgraded earlier models). The format for use with relative files is:

```
DOPEN # lfn,'filename",Ddrno, Lrlen
```

The logical file number lfn may, as usual, have any value from 1 to 255; drno, the drive number, must be 0 or 1; rlen is the length of each record in bytes. The record length may take any value from 1 byte upwards, provided that the overall size of the file does not exceed 182,880 bytes (i.e. 720 254-byte blocks)

```
DOPEN#1,"INCOME TAX",D1,L200
```

would thus open on drive 1 a file named "INCOME TAX", with a length of 200 bytes for each record on the file. Future operations on the file would then be carried out using the logical file number – in this case 1.

Note that at this point, although the length of each individual record within the file has been specified, the total number of records to be contained in the file has not. The initialisation process sets the length of the file, but before we can examine this process we must first consider the RECORD# command. Using this we may specify any desired number of records up to 64K (65,536).

RECORD # –. The file pointer. When writing to or reading from a random-access file, it is necessary to be able to specify the precise point in the file at which the next input or output operation is to begin. Both the record which is to be accessed, and the point within the record (i.e. the character within the record) at which the operation is to start, must be specified. This is done using a "file pointer" – an indicator which "points" at the appropriate character in the relevant record. The file pointer is positioned using the RECORD# command. The complete syntax of this command is

```
RECORD # lfn,recno,charno
```

lfn is the logical file number assigned in the DOPEN# statement, recno is the record number, which may have any value between 1 and 65,535 (i.e. 64K), and charno is the position of the character within the record to which the pointer is to be set. charno may have any value from 1 to 254. If no value is entered, charno is taken as 1 (the "default value").

Thus, if we wish to set the file pointer to the first character in the sixtieth record of a file which has been opened under the logical file number 7, the command will be:

```
RECORD#7,60,1
```

or just

```
RECORD#7,60
```

If we wish to set the pointer to the 27th character of the same record, the command will be:

RECORD#7,60,27

A useful modification of the RECORD# command allows us to access different records under program control. The command

RECORD#7,(N),39

will set the file pointer to the 39th character of the Nth record in the file, where N is some variable whose value can be changed by the program. Any allowed variable name can be used here, but *note that the variable name must be enclosed in brackets*. Once the file has been opened, the number of records must be specified. This is done by positioning the file pointer to the first character position of the last record in the file, using the RECORD# command, and then writing into that record the character CHR$(255). When this is done, the DOS will detect that all the other records are empty, and automatically write CHR$(255) into each, so initialising every record. Once a file has been opened and initialised, we may go on to enter data. Alternatively, the file may be closed, leaving it available for data entry at some later time. This is done by means of the command

DCLOSE#lfn

Example 11.3

A basic, though incomplete, program to create a relative file called "RELFILE", containing 75 entries each of 60 bytes, and located on the disk in drive 1, would be:

```
100 DOPEN#1,"RELFILE",D0,L60
110 RECORD#1,75
120 PRINT#1,CHR$(255);
130 DCLOSE#1
140 END
```

Note the semicolon in line 120; as in program 1, this prevents a terminator (line-feed and carriage return) from being transmitted to the file.

This simple program will reserve space on the disk and in the directory for the file "RELFILE". All the entries will as yet be blank, however.

Example 11.4

No provision has so far been made for dealing with disk errors; a more complete program would therefore be as follows:

```
10 REM      CREATE1ARELFILE
11 REM      ***************
100 DOPEN#1,"RELFILE",D0,L60
110 GOSUB 500
120 RECORD#1,75
130 GOSUB 500
140 PRINT#1,CHR$(255);
150 GOSUB 500
160 DCLOSE#1
170 END
500 REM    DISK ERROR SUBROUTINE
510 IF DS<20 OR DS=50 THEN RETURN
520 PRINTDS$
530 DCLOSE#1
540 END
```

An error subroutine of this type should be incorporated in *all* programs which make use of disk files.

NOTE When RECORD# is used as part of the initialisation process, an error message will be displayed, pointing out that the record being accessed does not yet exist. This fact should occasion no surprise, and the message may safely be ignored under these circumstances. The number of the error message will be 50, and the program above therefore ignores any error message with this number.

11.6.2 Writing data into a relative file

We may write as many items of data as will fit into each record, by setting the file pointer to the correct character position within the record, and then writing in the data by means of a PRINT# statement.

Only strings or string variables can be written to a relative file – numerical variables or numbers must be converted to strings (using STR$) before being written into the file.

Note also that entries to different fields within a record must be made in strict sequence, since writing an item to a file causes the remaining space between the end of that data item and the end of the file to be overwritten with zeroes. It is therefore not possible to rewrite a single field (except the last) within each record; the complete record must be transferred to the PET, modified as necessary, and then rewritten to disk by means of the PRINT# instruction.

Example 11.5

Let us consider first of all the problem of entering a single record into an existing file. For convenience we shall use the existing file "RELFILE" created in example 11.4, with its seventy-five records, each capable of holding sixty bytes of data (i.e. 60 characters). Let us assume that the file is to contain, say, the names, ages and telephone numbers of various persons. We may then allocate say, 35 characters to the name, 3, including the terminator, to the age (few people being older than 99) and the remaining 22 characters to the telephone number – this should suffice for even the longest number which may be encountered.

Then, if we wish to enter a new record – say Marcia Smith, 26, 061-246-8026 – at record number 35, a simple program to do this would be:

```
10 REM          MOD1RELFILE
11 REM          ***********
12 REM
100 DOPEN#1,"RELFILE",D0
110 GOSUB 500
120 G$="MARCIA SMITH":A$="26":T$="061-246-8026"
130 REM                         SET FILE POINTER TO   CHARACTER #1 OF RECORD #35
140 RECORD#1,35:REM             DEFAULT  CHARACTER POSITION IS 1
150 GOSUB 500
160 PRINT#1,G$
170 GOSUB 500
180 RECORD#1,35,36:REM          SET FILE POINTER TO START OF NEXT FIELD
190 GOSUB 500
200 PRINT#1,A$
210 GOSUB 500
220 RECORD#1,35,39:REM          SET FILE POINTER TO START OF NEXT FIELD
225 GOSUB 500
230 PRINT#1,T$
240 GOSUB 500
250 DCLOSE#1
260 END
500 IF DS<20          THEN RETURN
510 PRINT DS$
520 DCLOSE#1
530 END
```

Example 11.6

This last program can clearly be developed, allowing us either to enter or to modify at will any specific record in the file. We can therefore extend the program to allow the entry in turn of every record in the above file.

```
10 REM       WRITE1RELFILE
11 REM       ************
12 REM
100 PRINT"□INSERT DISK IN DRIVE #0"
120 INPUT "■NUMBER OF ENTRIES ";S
130 DOPEN#1,"RELFILE",D0
140 GOSUB 500
150 FOR I=1 TO S
160 RECORD#1,(I)
170 GOSUB 500
180 PRINT"□NAME #"I"?":INPUT G$
185 PRINT#1,G$
190 GOSUB 500
200 RECORD#1,(I),36
210 GOSUB 500
220 INPUT"■AGE ";A$
222 PRINT#1,A$
225 GOSUB 500
230 RECORD#1,(I),39
240 GOSUB 500
245 INPUT "■TELEPHONE NUMBER ";T$
246 PRINT#1,T$
248 GOSUB 500
249 NEXT I
250 DCLOSE#1
260 END
500 IF DS<20           THEN RETURN
510 PRINT DS$
520 DCLOSE#1
530 END
```

Example 11.7

This shows how any entry may be modified at will.

```
10 REM        MOD2RELFILE
11 REM        ***********
12 REM
100 PRINT"□INSERT DISK IN DRIVE #0"
120 DOPEN#1,"RELFILE",D0
130 INPUT "■WHICH RECORD ";N
140 GOSUB 500
160 RECORD#1,(N)
170 GOSUB 500
180 INPUT "■NAME ";G$
185 PRINT#1,G$
```

```
190 GOSUB 500
200 RECORD#1,(N),36
210 GOSUB 500
220 INPUT"■AGE ";A$
222 PRINT#1,A$
225 GOSUB 500
230 RECORD#1,(N),39
240 GOSUB 500
245 INPUT "■TELEPHONE NUMBER ";T$
246 PRINT#1,T$
248 GOSUB 500
249 INPUT "■FINISHED ";R$:IF LEFT$(R$,1)<>"Y" THEN GOTO 130
250 DCLOSE#1
260 END
500 IF DS<20            THEN RETURN
510 PRINT DS$
520 DCLOSE#1
530 END
```

11.6.3 Reading data from a relative file

In the previous programs, each data input statement was terminated by a line feed. This is written to the file record, and is recognised as an acceptable terminator by an INPUT# command. We may thus read any record, or indeed any field, from the file at will. To open the file for reading, we may again use the DOPEN# command; alternatively, we may use the normal OPEN command which we have already seen used with sequential files in the previous section. The latter has the advantage that, as in the demonstration program ORDERPRINT, we can build up the command by concatenating a series of strings. This allows the file name and the drive to be specified from the keyboard, something which is not possible if DOPEN # is used. The program which follows – example 11.8 – shows this technique in use.

Example 11.8

```
1 REM        READ2RELFILE
2 REM        ************
3 REM
100 INPUT "■WHICH DRIVE HOLDS DISK";D$
110 INPUT "NAME OF FILE";N$
120 OPEN2,8,2,D$+":"+N$+",R,R"
125 GOSUB 500
130 PRINT"■WHICH RECORD (F TO FINISH)";
135 INPUT R$:R=VAL(R$):IF LEFT$(R$,1)="F" THEN GOTO 170
140 RECORD#2,(R)
145 GOSUB 500
150 INPUT#2,NAME$
155 GOSUB 500
156 RECORD#2,(R),36:GOSUB 500
157 INPUT#2,A$:GOSUB 500
158 RECORD#2,(R),39: GOSUB 500
159 INPUT#2,T$:GOSUB 500
160 PRINT"■"NAME$
```

```
162 PRINTA$,T$
165 GOTO 130
170 CLOSE2
180 END
490 REM READ ERROR CHANNEL
500 IF DS>=20 THEN PRINTDS$:CLOSE2:END
510 RETURN
```

11.6.4 Searching a file

In this program – READ2RELFILE – only a single field of the specified record is accessed. In the example, this is the first field, but it would clearly be equally easy to access any other desired portion of each record. This is very useful when the file is to be searched with respect to some specific characteristic. For example, to continue with our 'little black book' program, we can very quickly search the file for persons of a specified age; their names and telephone numbers can then be listed on the printer. This searching process could just as well be carried out on a sequential file but the time required would, in the general case, be longer, since it would be necessary to read in the whole of each record in turn, rather than just a single (short) field, as here.

11.6.5 Indexing Random-Access Files

With a relatively short file such as that used in the exmples, it is quite easy to read through the whole file, checking only the required field of each record. With a very large file, however, the time required may become inconveniently large, and in such a case it will be more convenient to prepare an *index* – i.e. a list of file entries ordered according to some specified criterion. For instance, our demonstration file RELFILE could be indexed with respect to age, or even by location, as indicated by the STD code of the telephone numbers. Once an index has been prepared, it is then only necessary to look up the required characteristic in the index to obtain all the relevant record numbers.

11.7 WRITING AND READING SIMPLE USER FILES

With earlier machines, using BASIC 2.0, the process of writing and accessing direct-access files on the CBM disk system, although simple in concept, is in practice complicated by the number and nature of the commands which must be used. In the later systems extra facilities are provided which allow random-access files to be constructed much more simply than by the direct control of block allocation on the disk (see previous section). However, all the direct-access commands are available on the later systems, while on earlier (2000- and 3000-series) systems which have not been upgraded, there is no alternative to the use of direct-access or 'user' files. The handling of user files will therefore be discussed at some length.

It is usual, for simplicity, to make all the entries in a direct-access file *the same size,* although in general each entry will be divided into a number of 'fields' which will not normally be all of the same length.

For instance, in an address file, the name, address and telephone number might each occupy a separate field; clearly the field containing the address would normally be rather larger than that containing the telephone number. However, each field must be large enough to accommodate the longest name, or the longest address or telephone number, which is likely to be encountered. This being so, in most cases there will be a significant amount of waste space within each field. This does not occur with a sequential file, since there the length of each entry is known as soon as the entry is complete, and no space need be wasted.

Like a relative file, a user file therefore takes up more space than a sequential file containing the same data. On the other hand, an individual entry in a random-access file can be changed at will, whereas with a sequential file the whole file must be re-written.

To start with, let us consider the simplest possible case, where each entry consists only of a single item, and let us allocate one block to each entry. (This is usual, since it greatly simplifies programming, although a considerable amount of space may be wasted.) First, however, it will be necessary to examine the way in which data is stored in the CBM floppy disk system, since the creation and use of user files necessitates a detailed knowledge of the way in which disk space is allocated.

11.7.1 Storage of data on PET/CBM floppy disks

On the 2040, 3040 and 4040 CBM disk drive units, data and programs are stored on a number of

Figure 11.1 – see page 129

```
                    1         2
TRACK 01234567890123456789 0 USED FREE
01    ********************   00   21
02    ********************   00   21
03    ********************   00   21
04    ********************   00   21
05    ********************   00   21
06    ********************   00   21
07    ********************   00   21
08    ********************   00   21
09    ********************   00   21
10    ********************   00   21
11    ********************   00   21
12    ********************   00   21
13    ********************   00   21
14    ********************   00   21
15    ********************   00   21
16    ********************   00   21
17    ********************   00   21
18    **.****************    02   18
19    ********************   00   20
20    ********************   00   20
21    ********************   00   20
22    ********************   00   20
23    ********************   00   20
24    ********************   00   20
25    ******************     00   18
26    ******************     00   18
27    ******************     00   18
28    ******************     00   18
29    ******************     00   18
30    ******************     00   18
31    ****************       00   17
32    ****************       00   17
33    ****************       00   17
34    ****************       00   17
35    ****************       00   17

BLOCKS USED  2 FREE  688
```

concentric circular tracks. There are 35 such tracks, numbered from 1 to 35. Each track is divided into sectors or "blocks", and each block can hold 255 bytes of data. The number of fixed-length blocks on a track naturally varies with the circumference of the particular track; the outermost track carries 21 sectors, numbered 0 to 20, while the innermost track can hold only 17 (numbered 0 to 16). The total number of blocks is 690, so that the total usable capacity of the disk is almost 180,000 bytes, or about 30,000 words of English text (the equivalent of a short novel). Several blocks are used by the operating system, however, so that the space available to the user is somewhat less.

The data structure of each block is quite complex. A single file, whether program, sequential,

Figure 11.2 – see page 129

```
                    1         2
TRACK 01234567890123456789 0 USED FREE
01    ....................   00   21
02    ....................   00   21
03    ....................   00   21
04    ....................   00   21
05    ....................   00   21
06    ....................   00   21
07    ....................   00   21
08    ....................   00   21
09    ....................   00   21
10    ....................   00   21
11    ....................   00   21
12    ....................   00   21
13    ....................   00   21
14    ....................   00   21
15    ....................   00   21
16    ....................   00   21
17    *.....*.*.*.......*.*  06   15
18    **..................   02   18
19    ....................   00   20
20    ....................   00   20
21    ....................   00   20
22    ....................   00   20
23    ....................   00   20
24    ....................   00   20
25    ..................     00   18
26    ..................     00   18
27    ..................     00   18
28    ..................     00   18
29    ..................     00   18
30    ..................     00   18
31    ..................     00   17
32    ..................     00   17
33    ..................     00   17
34    ..................     00   17
35    ..................     00   17

BLOCKS USED  8 FREE  682
```

relative or user, may fill many blocks, which need not necessarily be physically adjacent to one another on the disk. Each block therefore contains a two-byte block pointer, which specifies the location (i.e. the track and sector) of the next block in the file. A number of other items of information used by the DOS (Disk Operating System) are also stored, along with 256 bytes of data. The first byte (byte 0) of the 256 is however reserved to indicate the location of the last valid character within the block, so that only 255 bytes are available for data storage.

Track 18 of each disk (i.e. the track midway between the innermost and outermost tracks) carries a *system file*. Sector 0 of track 18 holds the "Block Availability Map" or BAM, which carries details of

---

Figure 11.3 – see page 129

```
                    1         2
TRACK 01234567890123456789 USED FREE
01    ...................    00   21
02    ...................    00   21
03    ...................    00   21
04    ...................    00   21
05    ...................    00   21
06    ...................    00   21
07    ...................    00   21
08    ...................    00   21
09    ...................    00   21
10    ...................    00   21
11    ...................    00   21
12    ...................    00   21
13    ...................    00   21
14    *********.**.***.*..*   16   05
15    ********************   21   00
16    ********************   21   00
17    ********************   21   00
18    **..*..*..*..*..*...   07   13
19    ********************   20   00
20    ********************   20   00
21    ********************   20   00
22    .....*..............   01   19
23    ...................    00   20
24    ...................    00   20
25    ...................    00   18
26    ...................    00   18
27    ...................    00   18
28    ...................    00   18
29    ...................    00   18
30    ...................    00   18
31    ...................    00   17
32    ...................    00   17
33    ...................    00   17
34    ...................    00   17
35    ...................    00   17

BLOCKS USED  147 FREE  543
```

which blocks on the disk are already occupied and which are available for use. This occupies the first 144 bytes of the block, the remainder being used to store the "directory header" – i.e. the disk name and ID. Blocks 1 and 2 are reserved for the file directory proper.

NB When a disk is initialised, the BAM is read into the memory of the disk controller; at every access the ID of the disk is compared with that stored in the disk unit. If the disk is changed for another with the same ID number, and the drive is not re-initialised, *the stored BAM will be incorrect.* Since the ID's are the same, the unit will not recognise that the disk has been changed, and will write data to the disk using the wrong BAM. This is clearly likely to lead to the hopeless corruption of some or all of the

---

Figure 11.4 – see page 129

```
                  1         2
TRACK 01234567890123456789 0 USED FREE
01    ....................    00   21
02    *****.*.*.*****.*.*.*   15   06
03    **********************  21   00
04    **********************  21   00
05    **********************  21   00
06    **********************  21   00
07    **********************  21   00
08    **********************  21   00
09    **********************  21   00
10    **********************  21   00
11    **********************  21   00
12    **********************  21   00
13    **********************  21   00
14    **********************  21   00
15    **********************  21   00
16    **********************  21   00
17    **********************  21   00
18    **..*..*..*..*.......   06   14
19    ********************    20   00
20    ********************    20   00
21    ********************    20   00
22    ********************    20   00
23    ********************    20   00
24    ********************    20   00
25    ******************      18   00
26    ******************      18   00
27    ******************      18   00
28    ******************      18   00
29    ******************      18   00
30    ******************      18   00
31    ****************        17   00
32    ****************        17   00
33    *.*.*.**..*.*.*..       08   09
34    ................        00   17
35    ................        00   17

BLOCKS USED  606 FREE  84
```

files on the disk. To avoid any possibility of this depressing occurrence, *every disk should be given a different ID.*

11.7.2 User files and the BAM: VALIDATE and COLLECT

The commands VALIDATE and COLLECT tidy up the system files, removing from the BAM any blocks which do not belong to any of the files entered in the directory. Since a direct-access or USR file has no associated file-name, these commands will free the blocks of such a file, indicating them in the BAM as available for use. This can be avoided by writing a routine to reinstate them in the BAM, but the simplest procedure is to keep direct-access files on a separate disk, which carries no program or sequential files. In general the commands VALIDATE and COLLECT should not be used on disks carrying direct-access files.

Fig 11.1 shows a schematic plot of the block availability map for a disk which has been NEWed; occupied blocks are indicated by an asterisk. Note that only two blocks are occupied – those which hold the start of the system file (on track 18).

Fig 11.2 shows the same disk after the DOS support program has been written to it, while Figs 11.3 and 11.4 show the BAM's for disks which are respectively partially and almost completely filled. Note that the tracks closest to that containing the system file are filled first, since this reduces the amount of head movement required, and so minimises the access time.

These figures were obtained using DISKMAP, an invaluable program due to Hampshire. (The CBM utility program VIEWBAM performs a similar function.)

Example 11.9

```
100 REM    ****************
110 REM    *R/A1A-WRITE V.3*
120 REM    *E.A.FLINN      *
130 REM    *NOVEMBER 1981  *
140 REM    ****************
150 REM
160 REM    OPEN A CHANNEL AND RESERVE FIRST AVAILABLE BUFFER
170 OPEN1,8,3,"#":REM I.E. CHOOSE ARBITRARY CHANNEL NUMBER EQUAL TO 3
180 C=3
190 REM
200 REM   OPEN COMMAND AND ERROR CHANNEL
210 OPEN15,8,15
220 REM
230 A$="THIS IS ENTRY #"
240 B$="":FOR I=0 TO 254:B$=B$+CHR$(0):NEXT I
250 INPUT"WHICH DRIVE HOLDS DISK ";D
260 T=1:INPUT"WHICH TRACK ■■■■■■■";T$:T=VAL(T$):IF T$="*" THEN GO TO 260
270 FOR S=0 TO 20
280 REM   ALLOCATE BLOCK
290 PRINT#15,"B-A";D;T;S
300 REM    SET BUFFER POINTER
310 PRINT#15,"B-P";C;1
320 REM    OVER-WRITE BUFFER WITH ZEROES-NOT NECESSARY-USED HERE FOR CLARITY
330 PRINT#1,B$
340 REM
350 PRINT#15,"B-P";C;1
360 REM   WRITE DATA TO BUFFER
370 PRINT#1,A$+STR$(S);CHR$(13);
380 REM   WRITE BUFFER CONTENTS TO BLOCK
390 PRINT#15,"B-W";C;D;T;S
400 NEXT S
410 CLOSE1
420 CLOSE15
430 END
```

WRITING A SIMPLE DIRECT-ACCESS FILE

Let us consider, first in outline and then in more detail, the sequence of steps which must be carried out in order to write a random-access file directly to disk. The complete procedure is illustrated by the program of Example 11.9. For ease of understanding, each entry will consist only of a single item, and a separate block will be allocated to each entry.

1) A "channel" must be opened between the PET and the disk controller, and a section of memory – a 'buffer' – reserved within the disk controller. The term 'buffer' is used, since the use of this temporary store allows data to be input from the PET much faster than it can be written to disk. Each buffer consists of 256 bytes of memory – the contents of a block – although only 255 of these bytes can be used for data storage.

2) As in all programmed disk operations, a command channel must be opened between PET and disk controller, using a secondary address of 15.

3) The data or variables to be stored are written into the buffer. This can be done starting at any byte in the buffer, from 1 to 255 (byte 0 is reserved by the disk controller to indicate the location of the last valid character written into the buffer). The location of the first data character written into the buffer is controlled by setting a "buffer pointer". This simply specifies the byte within the buffer (from 1 to 255) at which the data will begin.

4) It is not absolutely essential to update the Block Availability Map to indicate that a specific block as been used, but it is a good idea, since an accurate BAM makes it easier to avoid over-writing existing data. The BAM is updated using the "block-allocate" instruction.

5) The data are next copied from the buffer into a specified block on the disk, freeing the buffer for re-use. The block is specified by means of the track and sector numbers.

6) Finally, if all disk operations are complete, the files are closed.

Let us now consider each of these steps in greater detail.

1) A channel is opened by a command of the form

   OPENlfn,8,cn,"#"

   lfn, the logical file number, can as usual take any value from 1 to 255 inclusive.

   The primary address – normally 8 – is that of the disk drive unit. The channel number, cn – the secondary address in the OPEN command – can have any value between 0 and 14. Values of 0 and 1 are, however, respectively interpreted as LOAD and SAVE commands, so that unless these operations are required, the channel number is restricted to the range 2-14. Up to five channels may be open at any one time.

   A typical command to open a buffer would thus be:

   OPEN1,8,3,"#"

   This will cause the first available buffer to be allocated to channel number 3, with an associated logical file number of 1.

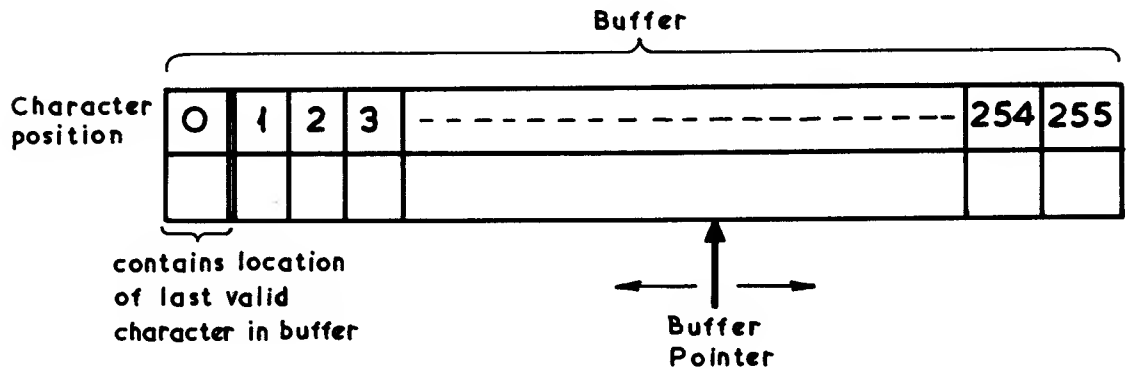
2) The command and error channel is opened by a simple OPEN instruction, using a secondary address of 15: for example

   OPEN15,8,15

   It is a convenient aid to memory here to use a logical file number of 15 also. The primary address is of course that of the disk drive.

3) The data or variables to be stored (only one in this case) are copied into the buffer. They can be written into any selected location, but in this case byte 1 will be the first used.

Figure 11.5



The buffer pointer is set via the command and error channel, using the "B-P" or "buffer-pointer" command. The syntax of this command is

PRINT#lfn, "B–P"; cn; cp

Thus, for example, if the logical file number 15 has been allocated to the command and error channel, and the channel number is 3, then the buffer pointer will be set to character position 1 (cp=1) by the command

PRINT#15, "B–P"; 3; 1

NB The syntax of this command may take a number of forms, all perfectly valid (see CBM Floppy Disk User Manual). It is however safer to stick to a single form, so as to reduce the risk of error.

4) The data or variables are then written to the buffer from the PET, using the command

PRINT#lfn, data

Thus, for instance, if we wish to copy the variable A$ into the buffer originally opened by the command

OPEN1, 8, 3, "#"

we use:

PRINT#1,A$

This will copy the string variable A$ into the buffer whose channel number is 3. The first character of A$ will be written at the location indicated by the buffer pointer, and the location of the last valid character will be stored in byte 0.

Terminators If the simple command above is used, the buffer will contain the string A$ followed by a carriage return *and* a line feed. Either of these on its own is adequate to denote the end of the string, and the second of these – the line feed – will therefore be read as the first character of any succeeding data. This can clearly cause trouble in some applications – e.g. when sorting. To avoid this, the transmission of the carriage return and line feed may be suppressed by means of a semicolon. A line feed on its own – CHR$(13) – is then sent, and a further semicolon is added to prevent the transmission of a further carriage return and line feed. The complete command is then:

PRINT#1,A$; CHR$(13);

5) The data are copied from the buffer into a specified block on the disk by means of the "block-write command". On unmodified 2040 and 3040 units, there are two allowable forms of this command, "B-W" and "U2". Only the second of these is available on 4040 and updated 2040/3040 units. The difference between the two commands is relatively slight. When "B-W" is used, the position of the last character written is stored in byte 0 of the block. This can, of course, be a serious disadvantage if only a part of the block contents is to be rewritten – e.g. when only one field of an entry is to be changed. When U2 is used, the contents of byte 0 are not altered, so that this command is more appropriate when a file is to

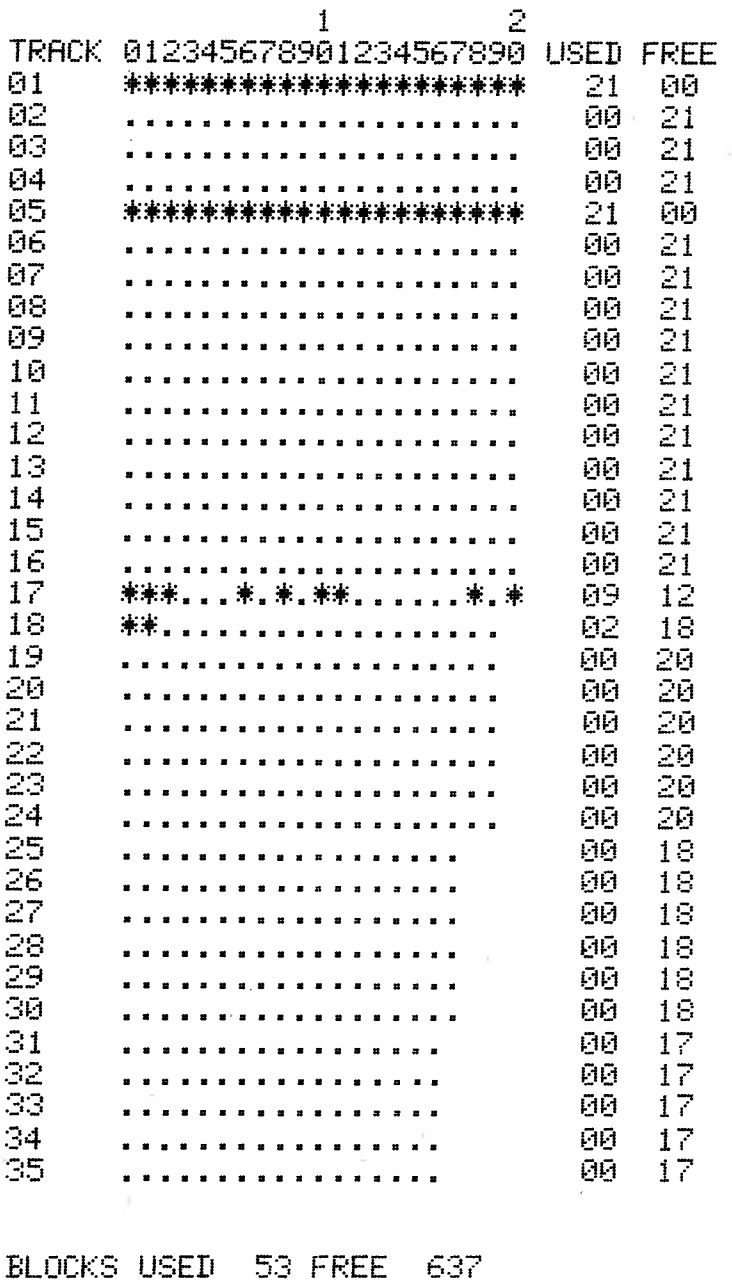


Figure 11.6

be up-dated, rather than written from scratch. Since "B-W" is not available on 4000-series systems, it is in general better to use U2 at all times, as this will make your programs more readily transferable between systems.

As with "B-P" there is a range of permissible syntaxes, but it is advisable to use one version consistently.

The format is

"B–W"; cn; drno; t; s

or "U2"; cn; drno; t; s

```
001:THIS IS     54  48  49  53  20  49  53  20
009:ENTRY #     45  4E  54  52  59  20  23  20
017:12←.....    31  32  0D  00  00  00  00  00
025:........    00  00  00  00  00  00  00  00
033:........    00  00  00  00  00  00  00  00
041:........    00  00  00  00  00  00  00  00
049:........    00  00  00  00  00  00  00  00
057:........    00  00  00  00  00  00  00  00
065:........    00  00  00  00  00  00  00  00
073:........    00  00  00  00  00  00  00  00
081:........    00  00  00  00  00  00  00  00
089:........    00  00  00  00  00  00  00  00
097:........    00  00  00  00  00  00  00  00
105:........    00  00  00  00  00  00  00  00
113:........    00  00  00  00  00  00  00  00
121:........    00  00  00  00  00  00  00  00
129:........    00  00  00  00  00  00  00  00
137:........    00  00  00  00  00  00  00  00
145:........    00  00  00  00  00  00  00  00
153:........    00  00  00  00  00  00  00  00
161:........    00  00  00  00  00  00  00  00
169:........    00  00  00  00  00  00  00  00
177:........    00  00  00  00  00  00  00  00
185:........    00  00  00  00  00  00  00  00
193:........    00  00  00  00  00  00  00  00
201:........    00  00  00  00  00  00  00  00
209:........    00  00  00  00  00  00  00  00
217:........    00  00  00  00  00  00  00  00
225:........    00  00  00  00  00  00  00  00
233:........    00  00  00  00  00  00  00  00
241:........    00  00  00  00  00  00  00  00
249:........    00  00  00  00  00  00  00  00

DRIVE  1  TRACK  10  SECTOR  12
```

Figure 11.7

where cn,drno,t and s are respectively the channel number, the number of the drive containing the disk, and the track and sector of the block to which the buffer is to be written. Using the channel-number 3, (as assumed in section 1), taking the disk to be in drive 0, and assuming that we wish to write to sector 13 of track 5, the appropriate command would then be (if the command and error channel has been given the logical file number 15):

PRINT#15, "B–W"; 3; 0; 5; 13

or PRINT#15, "U2"; 3; 0; 5; 13)

Now, bearing these points in mind, study Example 11.9 and see how they are applied in practice. This program writes into each sector S of some specified track T the message "THIS IS ENTRY #S". Since this very simple program expects to find 21 blocks on the track selected, only tracks 1 to 17 can be used. Fig 11.6 shows the Block Availability Map for a disk after the program of Example 11.9 has been run, using tracks 1 and 5. (The blocks on track 17 contain the DOS support program, while the two filled blocks on track 18 contain the disk header – see fig 11.2). The contents of all the 255 data bytes in a block may be examined by means of a special program such as SECTORPRINT (The PET Revealed by Nick Hampshire). This facility is invaluable when attempting to debug direct-access programs.

If program 11.9 is run without line 330, and the contents of each block are examined using SECTORPRINT or some similar program, it will be found in general that in addition to the expected message, a number of other characters are stored in the block, outside the region occupied by our string variable. When data are written to the buffer, any characters already present from previous usage of the buffer (for example, by the disk controller) are over-written; the region not over-written however remains unaffected. Then, when the buffer contents are written to disk, this unwanted

Example 11.10

```
100 REM     ***********
110 REM     *R/A1-READ*
120 REM     *EAF 1982 *
130 REM     ***********
140
150 INPUT"WHICH DRIVE CONTAINS DISK■■■■■■■■";D$:IF D$="*" THEN 150
160 D=VAL(D$)
170
180 REM        OPEN CHANNEL AND RESERVE FIRST AVAILABLE BUFFER
190 C=5:REM    SELECT CHANNEL NUMBER - HERE #5
200 OPEN1,8,5,"#"
210
220 REM        OPEN COMMAND AND ERROR CHANNEL (SA=15)
230 OPEN15,8,15
240
250 REM        SELECT TRACK
260 T=1:INPUT"WHICH TRACK■■■■■■■■";T$:T=VAL(T$):IF T$="*" THEN GOTO 260
270
280 REM       SELECT ENTRY TO BE READ
290 PRINT"WHICH FILE ENTRY";:INPUT"■■■■■■■■";F$:IF F$="*" THEN280
300 F=VAL(F$)
310
320 REM       READ APPROPRIATE ENTRY INTO BUFFER
330 PRINT#15,"B-R";C;D;1;F
340
350 INPUT#1,E$
360 PRINT:PRINTE$:PRINT
370 INPUT"FINISH - Y OR N■■■■■■■■";R$
380 IF LEFT$(R$,1)="N" THEN GOTO 290
390 CLOSE1:CLOSE15
400 END
```

(though harmless) information is stored also. Line 330 in the demonstration program overwrites such unwanted information with zeros, to clarify the process, but this is not necessary in a practical program. A print-out of the contents of a block, written using the program of Example 11.9 and read by means of SECTORPRINT, is shown in fig 11.7.

READING A USER FILE

In general, the process of reading back data from a direct-access file is similar to that of writing a file. We must first open a command and error channel, and a channel to a reserved buffer in the disk controller. The contents of a specified block are transferred to the buffer using the block-read command ("B-R" or "U1"). The data may then be read from the buffer using INPUT# or GET#.

```
001:THIS IS     54  48  49  53  20  49  53  20
009:THE FIRS    54  48  45  20  46  49  52  53
017:T PART O    54  20  50  41  52  54  20  4F
025:F ENTRY     46  20  45  4E  54  52  59  20
033:# 8←....    23  20  38  0D  00  00  00  00
041:........    00  00  00  00  00  00  00  00
049:........    00  00  00  00  00  00  00  00
057:........    00  00  00  00  00  00  00  00
065:........    00  00  00  00  00  00  00  00
073:........    00  00  00  00  00  00  00  00
081:........    00  00  00  00  00  00  00  00
089:........    00  00  00  00  00  00  00  00
097:....THIS    00  00  00  00  54  48  49  53
105: IS THE     20  49  53  20  54  48  45  20
113:SECOND P    53  45  43  4F  4E  44  20  50
121:ART OF E    41  52  54  20  4F  46  20  45
129:NTRY # 8    4E  54  52  59  20  23  20  38
137:←.......    0D  00  00  00  00  00  00  00
145:........    00  00  00  00  00  00  00  00
153:........    00  00  00  00  00  00  00  00
161:........    00  00  00  00  00  00  00  00
169:........    00  00  00  00  00  00  00  00
177:........    00  00  00  00  00  00  00  00
185:........    00  00  00  00  00  00  00  00
193:........    00  00  00  00  00  00  00  00
201:........    00  00  00  00  00  00  00  00
209:........    00  00  00  00  00  00  00  00
217:........    00  00  00  00  00  00  00  00
225:........    00  00  00  00  00  00  00  00
233:........    00  00  00  00  00  00  00  00
241:........    00  00  00  00  00  00  00  00
249:........    00  00  00  00  00  00  00  00

DRIVE  1  TRACK  15  SECTOR  8
```

Figure 11.8

The format of the block-read command is the same as that for block-write:

"B–R";cn;drno;t;s

or

"U1";cn;drno;t;s

The difference between the two is that "U1" forces byte 0 to 255. Thus, every character in the buffer can be read using INPUT# or GET#; "B-R" however leaves the position of the last valid character in byte 0 so that an INPUT# will terminate when it reaches this character.

This program demonstrates the process, and allows us to read back the (rather uninteresting) messages stored by the previous program.

```
001:THIS IS     54  48  49  53  20  49  53  20
009:THE FIRS    54  48  45  20  46  49  52  53
017:T PART 0    54  20  50  41  52  54  20  4F
025:F ENTRY     46  20  45  4E  54  52  59  20
033:# 14←...    23  20  31  34  0D  00  00  00
041:........    00  00  00  00  00  00  00  00
049:........    00  00  00  00  00  00  00  00
057:........    00  00  00  00  00  00  00  00
065:........    00  00  00  00  00  00  00  00
073:........    00  00  00  00  00  00  00  00
081:........    00  00  00  00  00  00  00  00
089:........    00  00  00  00  00  00  00  00
097:....THIS    00  00  00  00  54  48  49  53
105: IS A RE    20  49  53  20  41  20  52  45
113:VISED DA    56  49  53  45  44  20  44  41
121:TA ENTRY    54  41  20  45  4E  54  52  59
129: IN THE     20  49  4E  20  54  48  45  20
137:SECOND F    53  45  43  4F  4E  44  20  46
145:IELD←←..    49  45  4C  44  0D  0D  00  00
153:........    00  00  00  00  00  00  00  00
161:........    00  00  00  00  00  00  00  00
169:........    00  00  00  00  00  00  00  00
177:........    00  00  00  00  00  00  00  00
185:........    00  00  00  00  00  00  00  00
193:........    00  00  00  00  00  00  00  00
201:........    00  00  00  00  00  00  00  00
209:........    00  00  00  00  00  00  00  00
217:........    00  00  00  00  00  00  00  00
225:........    00  00  00  00  00  00  00  00
233:........    00  00  00  00  00  00  00  00
241:........    00  00  00  00  00  00  00  00
249:........    00  00  00  00  00  00  00  00


DRIVE  1  TRACK  15  SECTOR  14
```

Figure 11.9

Example 11.11

```
100 REM     ************
110 REM     *R/A2-WRITE*
120 REM     *EAF 1982  *
130 REM     ************
140
150 REM       THIS PROGRAM PRINTS TWO VARIABLES INTO EACH BLOCK
160
170 OPEN1,8,5,"#"
180 OPEN15,8,15
190 A$="THIS IS THE FIRST PART OF ENTRY #"
200 B$="THIS IS THE SECOND PART OF ENTRY #"
210 Z$="":FOR I=1 TO 254:Z$=Z$+CHR$(0):NEXT I
220 INPUT"WHICH DRIVE HOLDS DISK";D
230 T=1:INPUT "WHICH TRACK";T
240 FOR S=1 TO 20
250 PRINT#15,"B-A";5;D;T;S
260 PRINT#15,"B-P";5;1
270
280 PRINT#1,Z$:REM       OVER-WRITE BUFFER WITH ZEROES FOR CLARITY
285
290 PRINT#15,"B-P";5;1
300 REM       WRITE DATA TO BUFFER
310 PRINT#1,A$+STR$(S);CHR$(13);
320 PRINT#15,"B-P";5;101:REM     MOVE TO START OF NEXT FIELD
330 PRINT#1,B$+STR$(S);CHR$(13);
340 PRINT#15,"B-W";5;D;T;S
350 NEXT S
360 CLOSE1:CLOSE15
370 END
```

FILE ENTRIES WITH MORE THAN ONE FIELD

Where several items of data are to be input for each file entry, we must decide how the available space is to be divided between the different data fields. Assuming once more that a single block is allocated to each file entry, the available 255 bytes must be divided appropriately. In *Example 11.11*, two string variables are written to disk for each file entry. The first is allocated bytes 1-100, while the second is allocated bytes 101-255. As in program 11.9 the buffer is first filled with zeros, in the interests of clarity (line 280). The buffer pointer is set to character position 1 in line 290, and the first string (A$) is written in (line 310). The buffer pointer is set to byte 101 (line 320), and the second string is entered. The whole of the buffer is then written into the appropriate block using "B-W" or "U1" (line 340).

Fig 11.8 shows the result of running R/A2-WRITE. Provided that the location of each field is known, the data may be read out by an essentially similar process, as shown in program 11.12.

Alternatively, part of each record may be modified as in example 11.13. The result of running this program (with a different message) is shown in Fig 11.9.

NB In these demonstration programs illustrating the construction and use of user files, disk error routines have been omitted in the interests of clarity. In a program intended for serious application, the disk error channel would of course be checked after every disk operation.

In general, the availability of later versions of PET BASIC and DOS, which allow the use of relative files, has made the user file effectively obsolete. For the owners of earlier machines which have not been up-dated, however, the user file is essential wherever a random-access or direct-access file is required. The mechanism of creating and accessing a user file is in any case (besides being good

for the soul) a valuable study for those interested in the details of the processes by means of which computers store data. In the later CBM machines, although the same processes must be carried out, this is done by routines built into the software, which are transparent to the user.

Example 11.12

```
100 REM     ************
110 REM     *R/A2-READ*
120 REM     *EAF 1982 *
130 REM     ************
140
150 REM       THIS PROGRAM READS TWO STRING VARIABLES FROM EACH BLOCK
160
170 OPEN1,8,5,"#":REM RESERVE BUFFER - LABEL CHANNEL (#5)
180 OPEN15,8,15
210 INPUT "WHICH DRIVE HOLDS DISK";D
220 T=1:INPUT "WHICH TRACK";T
230 REM
240 REM       SELECT ENTRY TO BE READ
250 INPUT"WHICH FILE ENTRY DO YOU WANT TO SEE";F
260 REM
270 REM       READ APPROPRIATE ENTRY INTO BUFFER
280 PRINT#15,"U1:";5;D;T;F
290
300 REM READ AND DISPLAY BUFFER CONTENTS
310 INPUT#1,A$:PRINT:PRINT A$
320 PRINT#15,"B-P";5;101
330 INPUT#1,B$:PRINT:PRINT B$
340 INPUT"FINISH - Y OR N";R$
350 IF LEFT$(R$,1)="N" THEN GOTO 250
360 CLOSE1:CLOSE15
370 END
```

Example 11.13

```
100 REM     **************
110 REM     *R/A2-MODIFY*
120 REM     *EAF 1982    *
130 REM     **************
140
150 REM       THIS PROGRAM UPDATES THE SECOND FIELD IN EACH BLOCK
160
170 OPEN1,8,5,"#"
180 OPEN15,8,15
210 INPUT "WHICH DRIVE HOLDS DISK";D
220 T=1:INPUT "WHICH TRACK";T
225 INPUT "NEW ENTRY";A$
230 FOR S=1 TO 20
240 PRINT#15,"B-R";5;D;T;S
250 PRINT#15,"B-P";5;101
260 PRINT#1,A$;CHR$(13);
290 PRINT#15,"B-W";5;D;T;S
300 NEXT S
310 CLOSE1:CLOSE15
320 END
```

CHAPTER 12

Making Programs Crash-Proof and Friendly

12 Making Programs Crash-Proof and Friendly

Many computer programs – including expensive commercial packages – make no allowance for the fact that the user may well be someone with little or no technical knowledge, and no familiarity with computers. Thus, for instance, in word-processor packages the user – normally a typist with no computer background – is typically called on to remember a long list of arbitrary symbols before he or she can make use of the various facilities offered by these otherwise excellent programs. There is no excuse for this, and anyone designing programs or packages for general use should devote considerable effort to making them as "user-friendly" as possible.

With this end in view, the programmer should ensure that at all stages the prompts offered to the user are intelligible and helpful, and that the display itself is well laid-out, and easy to read and understand. Computer jargon should be avoided in the instructions and prompt messages. These should be written in good plain English and suitable for translation if used abroad. Moreover, where input is required from the user the program should be carefully structured, first to minimise the risk of an incorrect input, and then to ensure that a faulty input does not cause the program to fail.

A little thought at the design stage can save a great deal of subsequent ill-feeling and wasted time. Any program which "crashes" frequently, especially if this occurs after the user has already devoted considerable time to inputting data, will rapidly become very unpopular, and bring down considerable abuse upon the head of its designer. The programmer should therefore always strive to produce 'robust' programs. Robustness can be improved by the use of a few simple techniques, together with forethought and common sense.

Various common errors or misunderstandings on the part of the user may occur:

1) The user may return alpha-numeric data where the program requires a numeric input (e.g. "TEN" instead of "10"). This will cause the rather misleading message "REDO FROM START" to be output, striking terror into the heart of the inexpert. To avoid this difficulty *ALL inputs should be read in as strings,* regardless of whether the input data required are numeric or alpha-numeric (i.e. text).

   Once inside the machine, the input string can be dissected and inspected, using the techniques described in chapter 4. Where a numerical input is required by the program, the numerical value of the string can be extracted by means of the VAL function. Some care is necessary in this case, however, since the function VAL will return a value of zero in several cases where no valid numerical symbol has been input. This will occur wherever the first non-blank character of the string is not a numeric digit.

   Thus for example, the commands

   ```
   N = VAL("0")
   N = VAL("0AB2D7")
   N = VAL("TEN")
   ```

   will *all* result in a value of zero being assigned to the variable N.

Where zero is not a permitted value for N this will of course cause no difficulty. Otherwise, when a value of zero is returned, we must check the first character of the string, using the LEFT$ function.

2) The operator may accidentally have entered a valid but incorrect input. To check this, he should be given an opportunity to confirm the correctness of the input before it is fed into the program. This may be done immediately subsequent to each data entry, as in the demonstration program which follows. Alternatively, all the input data may be presented as a block, so that the user can alter any incorrect items. This necessitates a more complex program however.

3) If the operator accidentally or mischievously responds to a request for input data by pressing the RETURN key without pressing any other, execution of the program will be instantly terminated, and the PET will return to immediate mode. This may cause considerable inconvenience if the user has already worked part-way through a lengthy program. This can be avoided by inputting data *via a logical file*. A file is opened to the keyboard. Input data are then transferred to the logical file inside the machine, rather than being returned to the program, and a null entry will no longer cause the program to crash. This technique is seen in the input subroutine which starts at line 1920 of program example 12.1. A simple alternative method of avoiding null inputs, due originally to Hampshire, is demonstrated in program 12.2. This input routine can easily be defeated by a sufficiently malevolent or incompetent user, but it is very useful in programs where the highest degree of robustness is not essential.

4) Before an input is used in the program, it should be checked for validity – is it too large, too small, the wrong data type? If so, an appropriate message may be displayed, pointing out the error and requesting the user to supply more appropriate data. (See for example, lines 1520, 2000 and 2080 of program 12.1.)

5) If the RUN/STOP key is accidentally pressed while a program is running, the program will terminate. This may be prevented by disabling the stop key at the start of the program. However, the user should be offered from time to time an opportunity to terminate the program without turning off the machine.

On 3000 and 4000 series PET's the RUN/STOP key is controlled by the contents of memory location 144. (This location normally contains the decimal number 46 on 3000-series, and 85 on 4000-series machines.)

The key may be disabled by POKEing 49 or 88 respectively into this memory location, *but DON'T forget to re-enable the STOP key at the end of your program*. (See lines 1250 and 1880).

Where a Superchip or similar extension ROM is in use, the normal contents of location 144 are altered, and the operation of the machine will be disrupted if the program crashes while this location contains 49 (or 88). In this case, the machine must be turned off and on again to restore normal operation. This may be avoided by turning off the Superchip under program control at the start of the program. If this is not done, the initial contents of location 144 should be PEEKed and saved for restoration at the end of the program.

The demonstration program ELAMBDA listed below offers examples of the techniques described here, together with a demonstration of some of the facilities offered by the PET printer. The actual function of the program, which is somewhat technical, is unimportant; the program is used here simply as a vehicle to demonstrate useful techniques for interactive programs.

Example 12.1

```
1000 REM         ************************************************************
1010 REM         *ELAMBDA   V5.4                                             *
1020 REM         *A DEMONSTRATION PROGRAM TO ILLUSTRATE DESIRABLE FEATURES FOR*
1030 REM         *INTERACTIVE PROGRAMS                                       *
1040 REM         *E.A.FLINN       APRIL 1982                                 *
1050 REM         ************************************************************
1060 REM
1070 REM          THIS PROGRAM ALLOWS THE USER TO CHANGE AN INCORRECT INPUT
1080 REM
1090 REM          IT IS ALSO FULLY COMMENTED
1100 REM
1110 REM          OPEN PRINTER CHANNEL FOR TRANSMISSION OF DATA TO BE FORMATTED
1120 OPEN1,4,1
1130 REM          OPEN CHANNEL FOR TRANSMISSION OF FORMATTING DATA
1140 OPEN2,4,2
1150 REM          OPEN CHANNEL TO SCREEN (AS DEMONSTRATION ONLY)
1160 OPEN3,3
1170 REM          OPEN CHANNEL FOR PRINTING OF UNFORMATTED DATA
1180 OPEN4,4
1190 REM          OPEN CHANNEL TO SET LINE SPACING
1200 OPEN6,4,6
1210 REM          SET LINE SPACING
1220 PRINT#4
1230 PRINT#6,CHR$(36)
1240 REM          DISABLE STOP KEY
1250 POKE 144,PEEK(144)+3
1260 PRINT"□"
1270 PRINT#3
1280 PRINT#3
1290 PRINT#3,"     ELECTRON WAVELENGTHS"
1300 PRINT#3,"     ******** ***********"
1310 PRINT#3
1320 PRINT#3
1330 L$="LOWEST ENERGY IN ELECTRON-VOLTS"
1340 PRINT#3,L$;"?  ";
1350 GOSUB 1920
1360 IF R%=1 THEN GOTO 1310
1370 L=A
1380 H$="HIGHEST ENERGY IN ELECTRON-VOLTS"
1390 PRINT#3,"□"
1400 PRINT#3
1410 PRINT#3
1420 PRINT#3,L$;"=";L
1430 PRINT#3
1440 PRINT#3
1450 PRINT#3,H$;"? ";
1460 GOSUB 1920
1470 IF R% THEN GOTO 1380
1480 H=A
1490 IF L=<H THEN GOTO 1560
1500 PRINT
1510 PRINT
1520 PRINT"PLEASE DON'T BE SILLY!"
1530 FOR I=1 TO 2000:NEXT I
1540 PRINT"□"
1550 GOTO 1310
1560 S$="STEP SIZE IN ELECTRON-VOLTS"
1570 PRINT#3,"□"
1580 PRINT#3
1590 PRINT#3    - - -
1600 PRINT#3,L$;"=";L
1610 PRINT#3
1620 PRINT#3
1630 PRINT#3,H$;"=";H
1640 PRINT#3
1650 PRINT#3
```

```
1660 PRINT#3,S$;"?       ";
1670 GOSUB 1920
1680 IF R% THEN GOTO 1560
1690 S=A
1700 PRINT#4,"ENERGY(ELECTRON-VOLTS)      WAVELENGTH(ANGSTROMS)"
1710 PRINT#4
1720 REM          TRANSMIT FORMATTING DATA
1730 PRINT#2,"      999999                          ZZ.99
1740 K=12.1833
1750 FOR V=L TO H STEP S
1760 LA = (INT((100*K/SQR(V))+0.5))/100
1770 REM          TRANSMIT DATA FOR FORMATTING
1780 PRINT#1,V,LA
1790 NEXT V
1800 REM           RESET LINE SPACING
1810 PRINT#6,CHR$(24):PRINT#4
1820 CLOSE1
1830 CLOSE2
1840 CLOSE3
1850 CLOSE4
1860 CLOSE6
1870 REM          RE-ENABLE STOP KEY
1880 POKE 144,PEEK(144)-3
1890 END
1900 REM           INPUT SUBROUTINE
1910 REM           ***** **********
1920 R%=0
1930 REM          OPEN INPUT CHANNEL TO KEYBOARD
1940 OPEN5,0
1950 REM          INPUT VARIABLE AS STRING
1960 INPUT#5,A$
1970 REM          CHECK FOR NULL INPUT
1980 IF A$="" THEN R%=1
1990 REM          CHECK FOR NON-NUMERIC INPUT OR OUT-OF-RANGE VALUES
2000 W1$="INPUT MUST BE A FINITE POSITIVE NUMBER, EXPRESSED IN FIGURES"
2010 IF VAL(A$)>0 THEN GOTO 2070
2020 PRINT"□":PRINT:PRINT:PRINT
2030 PRINT W1$
2040 FOR J=1 TO 2000:NEXT J
2050 R%=1
2060 GOTO 2220
2070 IF VAL(A$)>0.1 AND VAL(A$)<1E6 THEN GOTO 2120
2080 W2$="NUMBER MUST BE GREATER THAN 0.1 AND LESS THAN 1000000"
2090 PRINT "□":PRINT:PRINT:PRINT
2100 PRINTW2$
2110 GOTO 2040
2120 PRINT:PRINT:PRINT
2130 PRINT TAB(18) VAL(A$)
2140 PRINT:PRINT
2150 PRINT"IS THAT CORRECT? PLEASE TYPE Y FOR YES, N FOR NO
2160 PRINT:PRINT
2170 INPUT#5,B$
2180 IF LEFT$(B$,1)="Y" THEN GOTO 2210
2190 R%=1
2200 REM           EXTRACT VALUE OF VARIABLE FROM INPUT STRING
2210 A=VAL(A$)
2220 CLOSE5
2230 RETURN
```

The program in fact produces a table of electron wavelengths corresponding to specified energies (don't worry about this if you don't understand it).

The user is asked to specify the lowest and highest energies of interest, and the interval between successive entries in the table. Once this information has been satisfactorily input, the wavelengths are calculated and printed out on the PET printer, using the built-in formatting facility.

The interest of the program, so far as we are concerned, lies in the techniques used to input the numbers required. Any invalid entry is rejected and the user is given a fresh opportunity to enter valid data. Once a valid input has been received, the user is given a chance to alter it before the program proceeds to the next step. The program starts by opening a number of channels to the printer (lines 1120 to 1200), so that the line spacing can be reset for improved legibility and the formatting facilities can be utilised. A channel is opened to the screen also to show that this can be done; it serves no other purpose in this program, although this facility is quite useful under some circumstances. The stop key is disabled (line 1250) and a title is displayed on the screen.

The user is then requested to input a value for the lowest energy in the table (line 1340). (Note that this prompt is produced by means of a PRINT statement, rather than an INPUT statement.) The program then jumps (line 1350) to the input subroutine (line 1920). This subroutine contains a number of points of interest.

First of all, the 'flag' variable R% is set to zero. So long as the user's input satisfies all the checks which follow as part of the subroutine, R% will remain zero, but should the input fail any of the checks, R% is set to 1. This is detected by the main program when we return from the input subroutine, and the appropriate prompt message is repeated. The input is read in as a string variable, A$, via a logical file which opens a channel to the keyboard. This ensures that the program will not crash if the RETURN key is pressed before an input has been entered. Should this occur, R% is duly set to 1 (line 2050) so that the prompt message will be repeated on returning to the main program.

The value of the number is extracted from the string A$ and checked to ensure that it is positive (line 2010). If not, an error message is displayed for a short period (lines 2030 and 2040) before the original prompt is repeated. The magnitude of the number is checked next (line 2070), to see that it lies within the specified limits (here 0.1 to 1,000,000). If not, a suitable error message is again displayed (line 2100) and R% is set to 1, so that the original prompt will be repeated.

Finally, if the number passes all these tests, it is displayed on the screen, and the user is given an opportunity to alter it (lines 2130-2180). If the user confirms that it is acceptable, the number is assigned as the value of the variable A, the input file is closed and we return to the main program, where the value of A is assigned to the appropriate variable – L, H or S. If the user rejects the input, the flag R% is set to 1, indicating an unacceptable input, and the original prompt is displayed once more, allowing the user to input a new number.

This process is repeated until a satisfactory set of numbers has been entered. The program checks (line 1490) to see that a humorous user does not enter an upper limit H which is higher than the lower limit L. However, the step size S is allowed to have any value between 0.1 and 1,000,000, regardless of the values of L and H. Even if S is greater than H, the program will run correctly, printing out a single entry which corresponds to the lower limit L. The wavelengths corresponding to the specified range of energies are then calculated and printed out (lines 1750-1790). The line spacing is reset to the standard value (line 1810), the various logical files are closed, and the RUN/STOP key is re-enabled – most important.

Note that line 1810 contains a plain unformatted PRINT# statement. This avoids problems if the program is run several times in succession. If more than one PRINT# statement with a secondary address of 6 is used in succession to alter the line spacing, it will be found that only the first such statement is obeyed. Just as in the case of the formatting facility (secondary address 2), instructions to alter the line spacing must be separated by ordinary unformatted PRINT# statements to overcome this bug – hence the second part of line 1810. (Try the effect of removing the PRINT#4 statement in line 1810 and running the program several times in succession.)

Program 12.2 demonstrates the simple alternative input routine mentioned earlier. The problems resulting from a null entry are avoided by ensuring that a valid character (here an asterisk) is always input, even when the user presses RETURN without making an entry. Although this routine is

not completely fool-proof, it is quick and easy to program, and has been used in several of the demonstration programs elsewhere in this book. Where a yes/no (Y/N) response is required, a "Y" (or an "N") can be entered as the default string, rather than an asterisk. In this case line 190 of course becomes superfluous.

```
100 REM    AN ALTERNATIVE INPUT ROUTINE
110 REM    ** ********** ***** *******
120 PRINT"□"
130 REM
140 INPUT"■■■■■■■";A$
150 REM
160 REM    THIS SHIFTS THE CURSOR THREE SPACES TO THE RIGHT,THEN PRINTS AN
170 REM  ASTERISK AND SHIFTS THE CURSOR THREE SPACES LEFT AGAIN
180 REM
190 IF A$="*" THEN PRINT"□";:GO TO 140
200 REM
210 REM    IF NOTHING HAS BEEN INPUT,A$ IS SIMPLY EQUAL TO "*"
220 REM   THE INPUT PROMPT IS THEREFORE REPEATED IN THE SAME SCREEN LOCATION
230 REM
240 PRINT"■■■■■■■■■■■■■"A$
250
260
270
280 REM   THIS ROUTINE CAN IF REQUIRED BE MODIFIED SO AS TO GIVE A VALID....
290 REM   ...DEFAULT REPLY TO THE PROMPT,BY PRINTING, FOR EXAMPLE, "Y" OR...
300 REM   ..."N" RATHER THAN "*"
```

CHAPTER 13

The PET/CBM In Control Applications

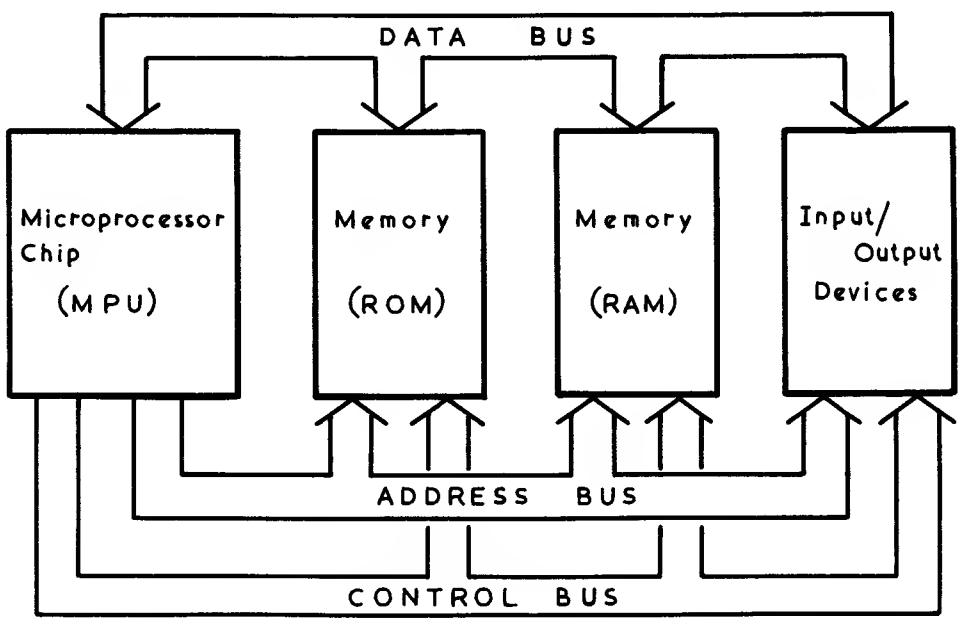
13 The PET/CBM In Control Applications

Most users think of desk-top computers simply as devices for handling data but in fact many hundreds of small computers like the PET are in daily use in industry, controlling and monitoring manufacturing processes or supervising testing procedures.

13.1 MICROPROCESSOR SYSTEM ARCHITECTURE

Before proceeding further it will be helpful to consider the basic layout or "architecture" of a microprocessor system, as shown schematically in Figure 13.1.

Figure 13.1



It will be seen that the microprocessor chip or MPU – the 6502 in the case of the PET/CBM – communicates with the memory and with any input/output devices by means of three main "data highways" or "buses". These are groups of parallel conductors along which digital signals may pass in either direction.

Most of the microprocessors currently in widespread use, like the 6502, have a data bus consisting of eight parallel conductors. Eight binary digits or "bits" of information – one byte – may therefore be sent simultaneously along the data bus.

Similarly, almost all of these "eight-bit" microprocessors have an address bus consisting of sixteen parallel conductors, so that sixteen bits of information can simultaneously be sent out on this bus. This information, of course, is sent in the form of electrical signals which are decoded to give an

"address'. This address specifies the precise location in memory from which data is to be read, or to which it is to be sent. Given sixteen bits, each of which can take on a value of either 0 or 1, there are 65,536 different and distinct addresses which can be sent out on the address bus. (This number is actually referred to as 64K in computer jargon; 1Kbyte = 2^{10} or 1,024 bytes.) The microprocessor can thus make use of memory up to a total capacity of 65,536 bytes of information, which is sufficient to store 65,536 alpha-numeric characters.

13.2 MEMORY-MAPPED INPUT/OUTPUT

In practice, it is relatively rare for a microcomputer to make use of all the available memory space, so that a large number of the possible memory addresses are in fact unused. This means that those combinations of address bits which do not refer to one of the memory locations actually in use can be decoded and used to activate specific input or output devices. Where this is done, so that the input/output devices occupy locations in the machine's "memory map" or "memory space", the system is said to use "memory-mapped input/output". Clearly, any microprocessor can use memory-mapped I/O, although some microprocessors (not the 6502) also have additional input-output facilities which allow the full 64K of memory to be used at the same time as a useful number of input/output devices.

The control bus in the microprocessor system carries the instructions sent out from the MPU to tell the other elements of the system precisely what is required. For example, if we wish to read a byte of information from a specific memory location, the appropriate address is sent out on the address bus and a "read" signal is sent out along the control bus. This causes the byte of data stored at the specified memory location to be sent out from the memory on the data bus in order that it can be read by the MPU. Similarly the MPU may send out a byte of data to be written into a specified memory location, a "write" signal being sent out on the control bus in this case, along with the appropriate address on the address bus.

In the PET, since memory-mapped I/O is used, it is possible to address an input or output device as if it were a portion of memory, data being written to or read from it in exactly the same way.

Essentially, therefore all input/output operations on the PET are identical with normal memory read/write operations. Although we can handle peripheral devices in just the same way as memory, PEEKing data from them, and POKEing data into them, we can, in some cases, make use of routines already present in the operating system of the PET which greatly simplify the process. We can input data from the keyboard and display it on the screen, or save it on disk or tape, without having to specify each step of the process in detail. In some control applications such as those which use the IEEE-488 interface this is true also, but in general we shall see that the programs for input/output applications must deal with data transfer in a very basic way.

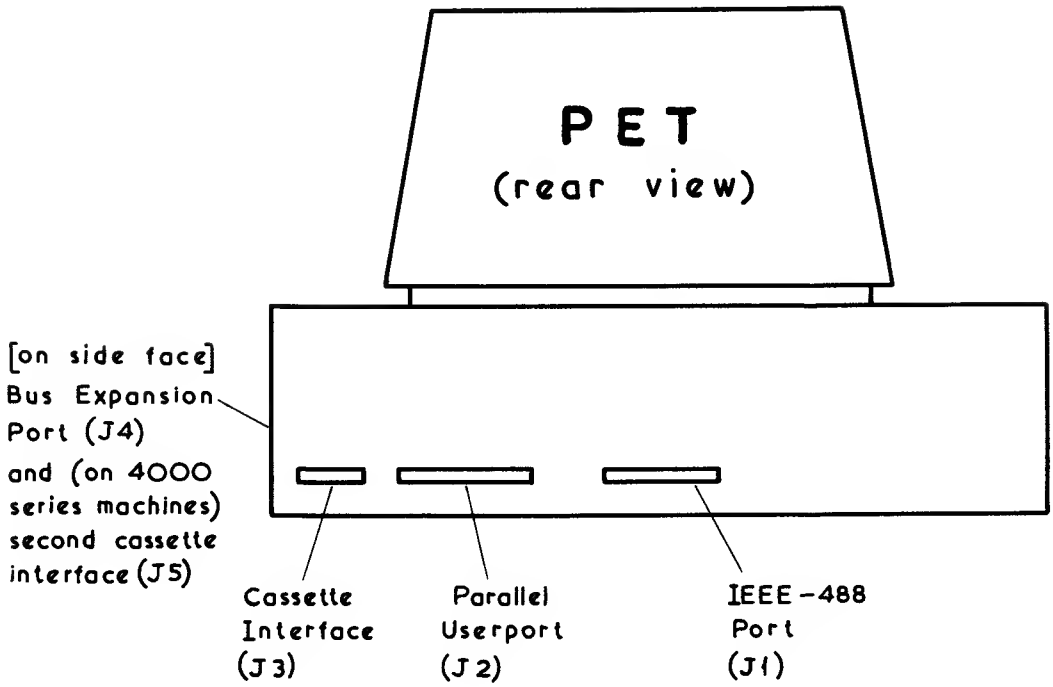
13.3 PET I/O INTERFACES

There are three different physical routes by which the PET can interact with external machines or instruments, other than the cassette deck; access to these is via edge connectors as shown in figure 13.2.

13.3.1 The IEEE-488 interface

This is routinely used to communicate with standard PET/CBM peripherals such as the printer or disk unit. The IEEE-488 bus system was initially devised for use in control and instrumentation applications however, so that the PET can very easily be used to monitor or control other instruments such as thermometers, digital voltmeters etc, if they are equipped with this industry-standard interface. The PET is in fact almost unique among desk-top computers in this respect, since on most machines the IEEE-488 interface is an expensive and often inconvenient extra, whereas with the PET it is standard equipment.

Figure 13.2



The PET operating system also contains a number of routines which greatly simplify the use of the IEEE-488 port from the programmer's view point.

Where the microcomputer must interact with devices not already equipped with an IEEE-488 interface, the construction of suitable interface circuits may be quite complex, so that the IEEE-488 port is not likely to be used under such circumstances.

13.3.2 The memory expansion port

Access to this is by means of a connector inside the machine. Since this port gives direct access to the control, data and address buses of the PET, and hence to the 6502 microprocessor which is the heart of the PET, this is a very flexible interface, allowing the PET to control processes or systems of considerable complexity. By the same token, however, it is equally possible to disrupt the operation of the PET, or to do major damage to its internal circuits if something goes wrong, and this port should be regarded as being strictly for the expert.

13.3.3 The user port

This is relatively easy to use, and like the IEEE-488 port interposes additional electronics between the main circuitry of the PET and the outside world. This makes it improbable that any serious damage can be done to the PET, although it is still possible that the special interface chip which controls the user port may be damaged by carelessness. The user port is the most likely of the PET's three main input/output ports to be of interest to the average user.

The PET user port is controlled by a special interface chip – the 6522 VIA, or Versatile Interface Adaptor. The 6522 carries out a number of functions and in fact is of comparable complexity to the 6502 microprocessor chip itself, having a total of 16 separately addressable registers. (A register is a store, capable of holding a binary number. In current microcomputers, most registers can hold an

eight-bit number.) The 6522 is connected to the address bus of the microcomputer in such a way that these registers occupy memory locations 59456-59471 inclusive. Quite apart from controlling the user port, the 6522 also carries out a number of other functions vital to the routine operation of the PET. Not all the registers are of interest in connection with the user port, therefore, and only those addresses considered to be of relevance are listed in Table 13.1.

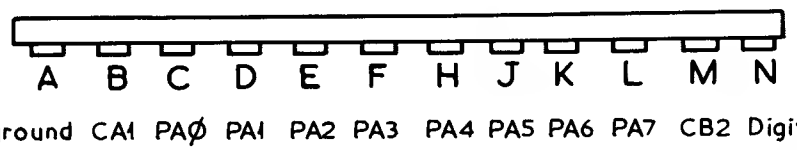
59457	($E841)	Port A data register (input/output) with handshake
59459	($E843)	Port A data direction register (DDR)
59464	($E848)	Timer 2 counter register (low-order byte)
59466	($E84A)	Shift register (SR)
59467	($E84B)	Auxiliary control Register (ACR)
59468	($E84C)	Peripheral control Register (PCR)
59469	($E84D)	Interrupt Flag Register (IFR)
59471	($E84F)	Port A data register – no handshake

Table 13.1
User port addresses

The significance of each of these registers will be explained in turn.

The user port is brought out to an edge connector, accessible from the rear of the PET (see figure 13.3). This connector is double-sided, and the user port connections are confined to the lower face of the connector. The connections on the upper face are intended primarily for diagnostic use in repair and maintenance, and should be left severely alone by any but the most knowledgeable user. A suitable socket to fit this edge connector is available from most CBM dealers.

Figure 13.3 User port connections as viewed from rear of PET



Pins A and N are return or "ground" connections for digital signals. Pins PA0-PA7 (C-L) are digital signal lines, each capable of carrying one bit of information at a time into or out of the PET (note that they are numbered from 0 to 7, rather than from 1 to 8; this is typical of computer usage). Pins CA1 and CB2 (B and M) are "handshake lines" or "control lines". Their prime function is to indicate either to the PET or to the external circuitry that a correct set of signals is present on PA0-PA7, and can now be read into or out of the computer. CB2 also has other functions, the most notable of which is probably its ability to generate audio signals, as demonstrated in numerous amusement arcade games.

The nomenclature PA0, PA1, CA1, CB2 and so on, may seem peculiar. In fact, the 6522 actually has two data "ports", Port A and Port B, each consisting of eight data lines, with one input control line C1 and one output control line C2. Not all these connections are brought out to the user port, so that lines PB0-PB7, CA2 and CB1 are not accessible.

NB Since the 6522 plays an important part in the internal workings of the PET, care should be taken when POKEing numbers into its registers. If an incorrect address is used, so that the number is written into the wrong register, the PET may refuse to respond to any further instructions. No permanent harm will result, but it can be very irritating to have to switch off the machine and start reloading your program from the beginning again.

13.4 DEMONSTRATION HARDWARE: DATA INPUT AND OUTPUT

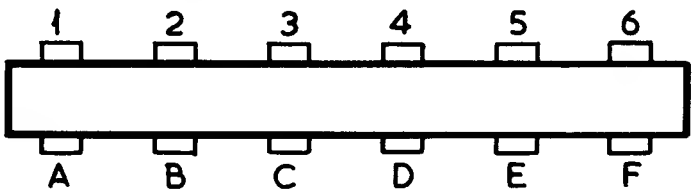
Special-purpose demonstration kits which clearly illustrate the operation of the user port can be obtained (for example, from A.R.B. Associates), and these are excellent if you can afford them. To start with, however, something cheaper may be considered adequate, and this can readily be constructed by anyone with a modicum of electronic experience.

Let us look first at a single data line, which can be set to act either as an input or an output line. When it is used as an input, we must be able to feed in either a logical 0 or a logical 1; when it acts as an output, we wish to know whether the bit sent out is a 0 or a 1.

In digital logic circuits, binary digits are indicated by different voltage levels. With the type of logic chips used in the PET, 5V indicates a one, and 0V a zero. We must therefore be able to input either 5V or 0V to a selected line on the user port; we must also be able to see whether a line is carrying a one or a zero, when it acts as either output or input. This is most easily done by means of a light-emitting diode or LED.

Fortunately, a stabilised 5V supply is readily available at the cassette interface. (see figure 13.4).

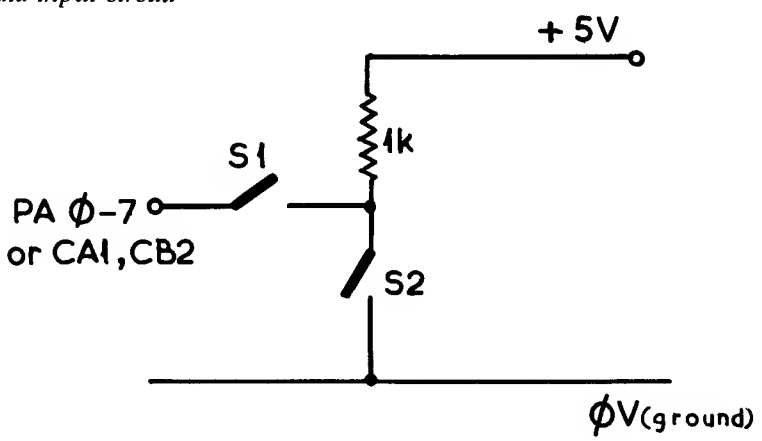
Figure 13.4 Cassette port connections as viewed from rear of PET.



On this connector, corresponding upper and lower pins such as pin A and pin 1 are electrically connected. A 5-V supply is readily taken from the two pins nearest the side face of the PET. (A and B or 1 and 2.)

WARNING: Pins 3 and C carry a 6-V unstabilised supply, and it is fatally easy to short this supply to the adjacent signal input pins D and 4, damaging the interface chip and rendering the cassette interface instantly unusable for loading programs. To avoid this, make sure that the power is off before connecting or disconnecting sockets at this edge connector. This risk is especially serious if you use any socket other than the rather expensive CBM product, so that it is worth tolerating the inconvenience of being unable to load or save programs on tape when you are drawing power from the cassette port, rather than trying to change plugs with the power on.

Figure 13.5 Data input circuit

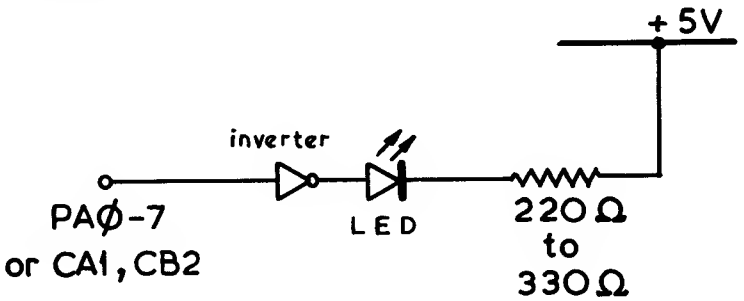


The simple circuit of Figure 13.5 allows either a 0 (S2 closed) or a 1 (S2 open) to be fed to one of the data or control lines when that line is used as an input. When the line is to be used as an output, S1 allows the data input circuit to be disconnected from the line.

Data indicator

Whether a user port line is being used for input or output, it is useful to have some visual indication of whether it carries a logical 1 or a 0. This is most conveniently done using a TTL inverter or inverting buffer, and any suitable LED.

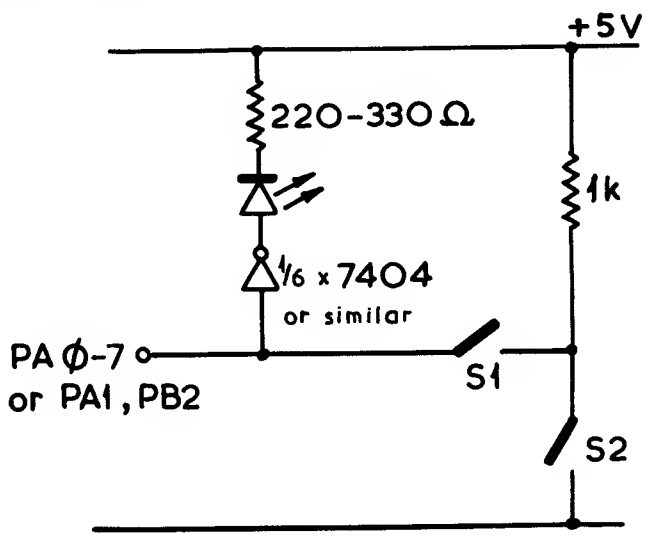
Figure 13.6 Data indicator



The inverter may be one section of a 7404, a TTL chip which carries six identical inverters. A 7404 can therefore be used to drive up to six LED's.

The complete input/indicator circuit for one line of the port then becomes

Figure 13.7 Complete input/indicator circuit



This may be multiplied if necessary (finances permitting) to give simultaneous indication and control on all ten of the port's lines. Care must be taken not to overload the 5-V cassette power supply, and it may be found advisable to increase the values of the resistors in series with the LED's to 330Ω, if ten such circuits – the maximum possible number – are to be used at once.

13.5 DATA INPUT/OUTPUT

13.5.1 The Data Direction Register

Each of the eight data lines PA0 to PA7 can be individually programmed to act either as an output or an input and furthermore can, if necessary, be repeatedly reprogrammed during the course of a program, so that signals can pass freely in either direction through the port. This programming is carried out by means of the data direction register (DDR). The DDR is essentially an 8-bit latch in which a one-byte number can be stored. (A latch is a circuit in which a binary number can be stored indefinitely). Each bit of the DDR corresponds to a specific data line.

If a specific bit in the DDR is set to 1, the corresponding data line will act as an output. If a bit is "reset", to 0, the corresponding data line will function as an input.

The DDR has the address 59459 (or $E843) in the PET's memory space, and the status of all the lines is simultaneously set by POKEing the appropriate number into this address.

Thus:

```
POKE59459,255
```

will cause each data line to act as an output, since 255 = 11111111 in binary.

Similarly,

```
POKE59459,0
```

will cause each data line to act as an input.

```
POKE59459,170
```

or

```
POKE59459,85
```

will cause alternate data lines to act as outputs, with the intervening lines acting as inputs, and so on. Note that bit 0 is the least significant bit – the extreme right-hand one – while bit 7 is the most significant (left-hand) bit.

13.5.2 The Data Register

Data to be sent out from the PET via the user port, or which has been input through the user port, is stored in two registers within the 6522 VIA. However, for all practical purposes these may be regarded as a single one-byte register, the Data Register or "DR", located at memory location 59471 ($E84F). A binary number may be POKEd into this location, and the DDR will ensure that only those bits corresponding to pins designated as outputs will be fed through to the appropriate pins of the user port. If location 59471 is PEEKed, the resulting number indicates the values of all the bits present on the user port pins, regardless of whether a particular pin is set to input or output. If the signals fed to the input pins change, the value of PEEK(59471) will change also. We may therefore determine whether any individual input pin is high or low by inspecting the corresponding bit of the one-byte number read from memory location 59471.

13.5.3 Simple input and output routines

Although we have so far seen only a small part of the user port's facilities, we are now in a position to write programs which will allow the user to carry out straightforward input and output. The same principles can also be put to good use in the control of simple experiments or models.

Program 13.1 sets all the user port pins to act as outputs. A number from 0 to 255, input from the

keyboard, is then output in binary form via the user port, setting each line of the user port high or low according to the value of the number input.

The program is easily modified to output the numbers from 0 to 255 in order, with a slight delay between numbers: the output lines then count up from 0 to 11111111 in binary (Example 13.2).

Example 13.1

```
100 REM         ***************
110 REM         *SIMPLE OUTPUT*
120 REM         *EAF 1982     *
130 REM         ***************
140
150 PRINT"■■■DON'T FORGET TO OPEN THE DATA ISOLATION SWITCHES"
160
170 POKE59459,255:REM      SET ALL USER PORT LINES TO OUTPUT
180 INPUT"■■TYPE IN A NUMBER (0-255)";N
190 POKE59471,N
200 GOTO 180
```

Example 13.2

```
100 REM       ******************
110 REM       *BINARY COUNTING*
120 REM       *EAF 1982       *
130 REM       ******************
140
150 PRINT"■DON'T FORGET TO ISOLATE DATA SWITCHES"
160
170 POKE59459,255:REM      SET ALL DATA LINES TO OUTPUT
180 FOR J=1 TO 255
190 POKE59471,J:REM        OUTPUT NUMBER J TO USERPORT
200 FOR K=0 TO 50:NEXT K:REM      DELAY
210 NEXT J
220 GOTO 180
```

The program of example 13.3 sets the four high-order bits of the user port to input, and the four low-order bits to output. The number input on the four high-order lines is shifted four places (i.e. divided by 16), and after a delay is output on the four low-order lines.

Example 13.3

```
100 REM         ***************
110 REM         *INPUT/OUTPUT*
120 REM         *EAF 1982     *
130 REM         ***************
140
150 PRINT"■OPEN 4 LOW-ORDER ISOLATING SWITCHES:CLOSE 4 HIGH-ORDER SWITCHES"
160 PRINT"■■THE VALUES OF THE 4 HIGH-ORDER DATA SWITCHES ARE REFLECTED";
170 PRINT" AFTER A DELAY BY THE 4 LOW-ORDER LED'S"
180
190 POKE59459,15:REM        4 LOW BITS OUTPUT:4 HIGH BITS INPUT
200 N%=PEEK(59471)
210 N1%=N% AND 240:REM      CLEAR 4 LOW-ORDER BITS
220 FOR J=1 TO 250:NEXT J:REM     DELAY
230 POKE59471,N1%/16:REM    SHIFT 4 PLACES RIGHT AND OUTPUT
240 GOTO 200
```

Program 13.4 sets all the lines to input, and displays on the screen the decimal value of the binary number fed in via the pins of the user port. (Note that if an input pin is left disconnected, the PET will respond as if a 1 had been input on that line.)

Example 13.4

```
100 REM       ***************
110 REM       *INPUT DISPLAY*
120 REM       *EAF 1982     *
130 REM       ***************
140
150 PRINT"■■■DON'T FORGET TO CLOSE THE DATA ISOLATION SWITCHES"
160
170 POKE59459,0:REM     SET ALL USER PORT LINES TO INPUT
180
190 PRINT"SET THE DATA SWITCHES AND THEN PRESS ANY KEY"
200 GET A$:IF A$="" THEN GOTO 200
210 N=PEEK(59471):REM     READ INPUT
220 PRINT"■■N="N
230 GOTO190
```

13.6 MASKING

In control applications it is often necessary to determine whether a specific input line is high or low (e.g. whether a switch, thermostat, relay, etc. is open or closed). Some microprocessors have facilities for checking or altering individual bits of a one-byte number or "word", but unfortunately these are not available on the PET's 6502 microprocessor. To determine or alter the value of a single bit read in from the user port it is therefore necessary to use a technique known as "masking". Masking relies on the use of the logical operations AND and OR, and while it would not be appropriate here to launch into a lengthy explanation of Boolean algebra, it is necessary to look a little more closely at these logical operations.

Without going into the origin of the terms, we can define the basic logical operations by means of "truth tables". In what follows a, b and c are logical variables, each of which can take only the values 0 or 1.

AND

If a = 1 and b = 1, a AND b = 1
If either a = 0 or b = 0 or both, a AND b = 0
Writing this in the form of a truth table, we have, putting c = a AND b, the following result.

a	b	c	
0	0	0	
1	0	0	a AND b
0	1	0	
1	1	1	

OR

If *either* a or b, or both, is equal to 1 then a OR b = 1. If both a and b are zero, a OR b = 0.

a	b	c	
0	0	0	
1	0	1	a OR b
0	1	1	
1	1	1	

Although strictly speaking the logical operations described above are valid only for one-bit numbers, the same terms are used in the microprocessor world to describe the process of carrying out the operation bit-by-bit on all the bits in two binary numbers of the same length.

Thus

```
          00001111
AND       11000011
          --------
          00000011
```

Only in bit positions 0 and 1 are the bits of both numbers equal to 1, so only bits 0 and 1 of the result will be equal to 1.

We can select out a single bit of a number by ANDing the number with a "mask" containing only a single 1:

```
          1011X011
AND       00001000
          --------
          0000X000
```

Here every bit except that indicated by X will be cleared to a 0; the value of that bit will be unaltered.

We may similarly clear a specified bit – i.e. reset it to 0. For instance, if we wish to reset bit 0 of a number to a zero, we can achieve this by ANDing the number with 254 (i.e. 11111110 in binary).

```
          1011011X
AND       11111110
          --------
          10110110
```

Unfortunately, the clarity of the process is obscured when the binary numbers are converted to their decimal equivalents.

For instance, although it may seem unlikely at first sight,

```
                    127 AND 129 = 1

          01111111    =  127
AND       10000001    =  129
          --------
          00000001    =    1
```

In spite of its superficial complexity, the technique of masking must be mastered if we are to use microprocessors or microcomputers in control applications.

We have seen that we can clear any desired bits – i.e. reset them to 0 – by ANDing a number with a suitable mask. We may also set a bit or bits to 1, using the logical operation OR; the remaining bits are unaffected.

For example 127 OR 128 = 255

```
          01111111    =  127
OR        10000000    =  128
          --------
          11111111    =  255
```

If we OR a number with a mask containing only a single 1, then the corresponding bit of the original number will be set to 1, regardless of its original value

```
          XXXXXXXX
OR        00001000
          --------
          XXXX1XXX
```

so that for example

```
          10101X01
OR        00000100
          --------
          10101101
```

Both these techniques are used in the programs which follow.

Example 13.5

This program shows how each bit in turn of a number read in from the user port may be checked.

```
100 REM         ***************
110 REM         *INPUT MASKING*
120 REM         *EAF 1982     *
130 REM         ***************
140
150 PRINT"□DON'T FORGET TO CLOSE THE DATA ISOLATION SWITCHES"
160 PRINT"THE NUMBER SET ON THE DATA SWITCHES IS READ IN AND PRINTED ON SCREEN"
170 FOR J=1 TO 2000:NEXT J
180
190 POKE59459,0:REM        SET ALL DATA LINES TO INPUT
200 N1=256:REM             ENSURES PRINT-OUT AT START
210 N=PEEK(59471):REM      READ INPUT
220 IF N1=N GOTO 210:REM REWRITE DISPLAY ONLY IF N CHANGES
230 PRINT"□N ="N
240
250 REM                    NOW CHECK EACH BIT IN TURN BY MASKING THE INPUT DATA
260
270 FOR J=7 TO 0 STEP -1
280 N%(J)=0
290 IF 2↑J AND N THEN N%(J)=1
300 PRINT"BIT"J" ="N%(J)
310 NEXT J
320 N1=N
330 GOTO 210
```

This program displays the value of each bit separately, changing the readout as the input signal varies.

13.7 THE HANDSHAKE LINES – CA1 AND CB2

So far we have not considered the significance of the control or "handshake" lines CA1 and CB2.

In the simple examples considered so far, the data presented at the input has been stable – i.e. it has not been changing rapidly. In a real control system this is often not the case – for instance the output from an analogue-to-digital converter may vary almost continuously. In a case like this, it is necessary to ensure that data should be accepted only when the signals presented to the user port are valid.

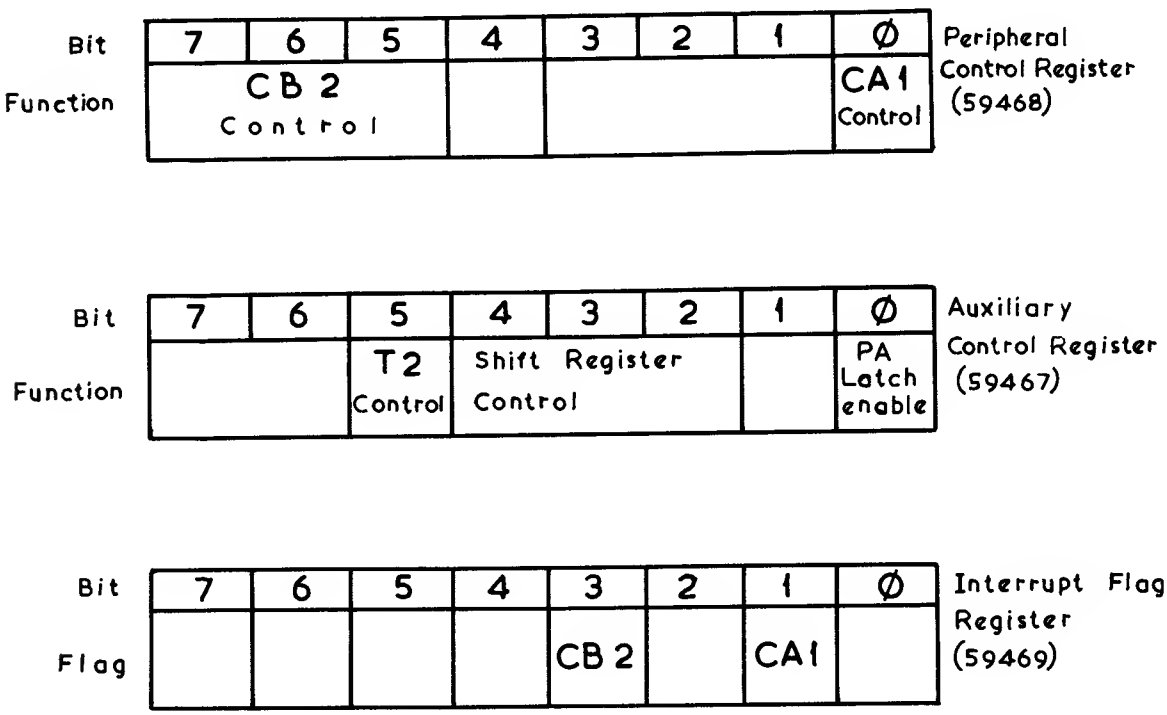
To achieve this, the "handshake data register" (HDR) is used. This lives at address 59457 ($E841). Data are latched into the HDR from the data register only when line CA1 detects a

transition from 0 to 1 or from 1 to 0. We can choose to trigger the latching process on either a positive-going or a negative-going transition.

When a pulse on CA1 causes data to be latched into the HDR, a 'flag' bit in yet another register (the 'interrupt flag register' or IFR) is set to 1. Until this occurs, the number in the HDR will change with that in the DR; once the flag bit is set, either by a latching pulse on CA1 or by a suitable POKE command, the contents of the HDR remain unaltered until they have been read (e.g. by PEEKing location 59457) or the flags reset to zero by a suitable POKE instruction. Reading the contents of the HDR clears the corresponding flag bit in the IFR and frees the HDR to follow the contents of the DR until the next latching or "handshake" pulse is received.

The latching process is controlled also by bits in two more registers of the 6522, the "peripheral control register" or PCR (memory location 59468) and the "auxiliary control register" or ACR (memory location 59467). These registers, along with the IFR, are shown in Figure 13.8 below.

Figure 13.8



N.B. In these registers, bits other than those marked control various internal operations within the PET. Great care should therefore be exercised when POKEing these locations.

It will be seen that bit 1 of the IFR acts as the flag which indicates whether or not data have been latched into the HDR. The value of this bit can therefore be checked by PEEKing location 59469 (the IFR) and ANDing the contents with 2 (00000010). If the result is non-zero then valid input data have been latched into the handshake data register (location 59457). We may alternatively use the WAIT instruction. This suspends operation of the machine until a specified bit or bits of a memory location go high. In the case of the CA1 flag in the Interrupt Flag Register, the appropriate instruction is

```
WAIT59469,2
```

However the WAIT instruction can be troublesome. Until the specified bit pattern is received, the machine will obey no further BASIC instructions and even the RUN/STOP key will not restart it. If, therefore, you make a mistake or the external circuit fails to deliver the appropriate signal, the only way to recover is to switch off the PET and start again, unless you have fitted a RESET switch to the machine.

The latching or "handshaking" mechanism is enabled or disabled by setting bit 0 of the Auxiliary Control Register (memory location 59467) to 1 or 0 respectively. Positive-edge or negative-edge triggering is selected by bit 0 of the Peripheral Control Register (PCR). If this is set to 0, the CA1 input will be triggered by a negative-going edge (Figure 13.9).

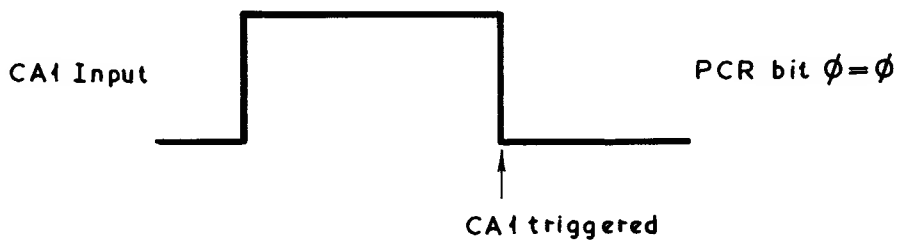


Figure 13.9

If bit 0 of the PCR is set to 1, CA1 will trigger on a positive edge . (Figure 13.10).

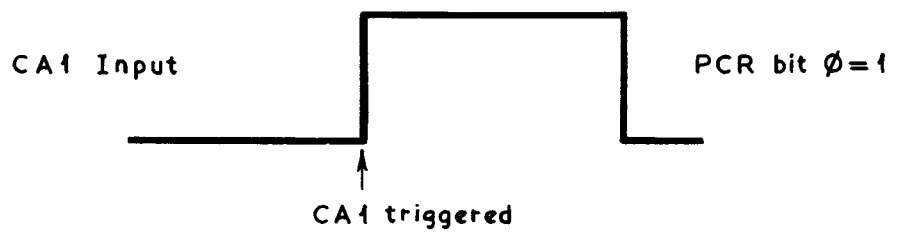


Figure 13.10

Example 13.6 shows how a signal on CA1 latches data into the handshake data register. The reader should confirm that latching does, in fact, occur only for the specified direction of transition on CA1 (but see notes on contact bounce).

Example 13.6

```
100 REM     HANDSHAKE-CA1
110 REM     ************
120
130 PRINT"■"
140 POKE 59467,PEEK(59467)OR1:REM      SET LATCHING
150 POKE 59468,PEEK(59468)OR1:REM      POSITIVE-EDGE TRIGGERING
160 INPUT "POSITIVE- OR NEGATIVE-EDGE TRIGGERING (P OR N)";R$
170 IF LEFT$(R$,1)="P" THEN GOTO 190
180 POKE 59468,PEEK(59468)AND 254:REM  NEGATIVE-EDGE TRIGGERING
190 PRINT"■CLOSE ALL DATA AND ISOLATING SWITCHES"
200 PRINT"■■ SET DATA SWITCHES TO DESIRED NUMBER"
210 PRINT"THEN PRESS CA1 SWITCH AND SET DATA      SWITCHES TO A NEW NUMBER"
220 PRINT"■■WHEN YOU HAVE DONE,THAT PRESS ANY KEY"
230
240 GET A$:IF A$="" THEN GOTO 240
```

```
250 F=0:IF PEEK(59469) AND 2 THEN F=1:REM  CHECK FLAG
260 L=PEEK(59457):U=PEEK(59471):REM    READ HDR AND DR
270 PRINT"■■FLAG BIT ="F
280 PRINT"HDR CONTENTS ="L
290 PRINT"DR  CONTENTS ="U
300 FOR I=1 TO 2000:NEXT I
310 PRINT"■■ONCE READ,THE REGISTER IS UNLATCHED:ON  RE-READING,"
320 F=0:IF PEEK(59469) AND 2 THEN F=1:REM  CHECK FLAG
330 L=PEEK(59457):U=PEEK(59471):REM    READ HDR AND DR
340 PRINT"■■FLAG BIT ="F
350 PRINT"HDR CONTENTS ="L
360 PRINT"DR  CONTENTS ="U
370 END
```

In general, it is unwise to alter the contents of the control registers, except for the specific bits detailed here. The VIA is an integral part of the PET, which may be brought to a state of complete immobility by incautious POKEing. Should this occur, the machine must be switched off to reset all the register contents to normal.

13.8 CONTACT BOUNCE

When a mechanical switch is closed, the contacts often bounce apart after initially making contact. Instead of applying a single voltage step to the attached circuit, the switch may thus apply several undesired pulses in rapid succession as the contacts bounce open and close again before the system settles down. There are a number of ways in which the effects of such contact bounce may be negated, but in microcomputer applications it is easiest to use "software debouncing".

The switch contacts normally bounce for no more than a few milliseconds, and all that is necessary is to introduce a short delay after the first closure of the switch is detected, before allowing a further signal to be accepted from that switch. For instance, program 13.7 will time the interval between successive input pulses on CA1.

Example 13.7

```
120 REM       CA1 INPUT TIMER
130 REM       ***************
140 PRINT"■PRESS CA1 FOR INPUT":N=0
150 :
160 POKE59468,PEEK(59468)OR1:REM SET CA1 TO DETECT +TIVE EDGE(SET PCR BIT0 TO1)
170 :
180 A=PEEK(59457): REM RESET CA1 FLAG IN IFR BY READING HDR
190 :
200 IFPEEK(59469)AND2THEN230:REM CHECK CA1 FLAG IN IFR,IF BIT1=1 THEN PROCEED
210 :
220 GOTO 200
230 P=1+P  :IFN=1 THEN 270
240 PRINT"■FIRST INPUT DETECTED ON CA1"
250 T=TI:N=1
260 GOTO180
270 PRINT"■SECOND INPUT DETECTED ON CA1"
280 T1=.01*INT(100*(TI-T)/60)
290 PRINT"■TIME BETWEEN INPUTS=";T1;"SECS"
300 PRINT"■NUMBER OF INPUTS=";P
310 IFT1<.2THEN PRINT"■CONTACT BOUNCE! TRY AGAIN
320 PRINT"■PRESS ANY KEY TO CONTINUE
330 GETA$:IFA$=""THEN 330
340 PRINT"■
350 GOTO 140
```

13.9 THE CB2 HANDSHAKE LINE

This can be used as either an input or an output; in the latter case, a number of additional facilities render it very versatile.

There are associated with the CB2 line a timer and a shift register (SR). The timer works by counting a specified number of cycles (up to 255) of the PET's 1-MHz master clock. When the appropriate number of clock cycles has passed, the timer sends a shift pulse to the shift register, and starts counting from zero once more.

A shift register is a register in which the contents can be *shifted* along at will on receipt of a suitable shift pulse. In the case of the 6522 VIA, it is possible to arrange matters so that as the contents are shifted to the left, the bit shifted out of the highest-order position is fed to the CB2 line, and simultaneously fed back into the lowest-order bit position (see figure 13.11). The eight bits initially loaded into the SR can therefore be cycled round and round continually, moving along 1 bit position every time the timer indicates the passage of a specified number of clock cycles, so that a corresponding regular cyclic 8-bit pattern is sent out on the CB2 line. The rate at which this signal is sent, and hence the frequency of the output signal on CB2, is under the control of the programmer, who can load into the timer any number between 1 and 255.

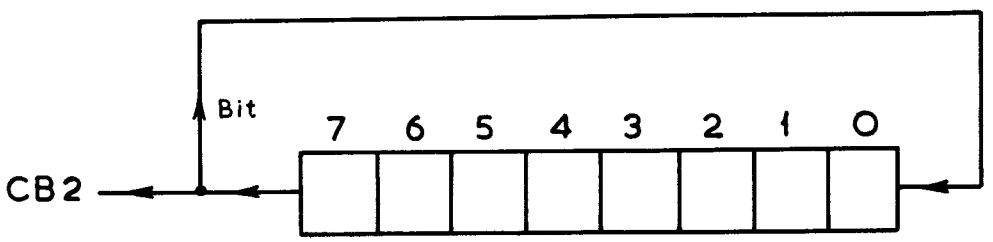


Figure 13.11

13.10 CB2 AS INPUT

When CB2 is to be used as an input line the shift register must first be disabled by clearing bits 2, 3 and 4 of the ACR (Auxiliary Control Register) to 0. This is done by the command

```
POKE59467,PEEK(59467)AND227
```

Bits 5 and 7 of the PCR must be set to 0, and bit 6 of the PCR must be set to 0 or 1, according to whether handshaking is to take place on negative or positive edges, respectively.

When CB2 is triggered, bit 3 of the IFR is set. However, it will be found that it is reset almost instantly, as port B is constantly read by the PET in its normal operation. A handshake pulse on CB2 is therefore best detected by the WAIT instruction; it will be found that the alternative PEEKing and masking procedure, which works well on CA1, cannot be used. Furthermore, the data register need not be PEEKed or the IFR POKEd to reset the flag, although this would normally be necessary if the PET were not itself carrying out the process continually.

In its input mode CB2 is most useful in *timing* applications. Successive pulses on CB2, or on CA1 and CB2, may be detected and the time interval between these events can then be determined either by means of the PET's internal clock (TI or TI$) or by means of a simple machine-code routine. This principle is the same as that demonstrated in Example 13.7.

13.11 CB2 AS OUTPUT: SOUND GENERATOR

When used as an output, CB2 can supply a reasonable amount of current – 1mA at 1.5V. In addition to its use as a handshake output, indicating that valid output data are present on port A, it can

therefore also be used to drive quite basic electronic devices, such as a simple sound box. This is particularly useful when used in connection with the shift-register and timer facilities.

As before, if CB2 is to function as a handshake line, the SR must be disabled by clearing bits 2, 3 and 4 of the Auxiliary Control Register. Bits 6 and 7 of the PCR are set to 1. The signal on CB2 is then controlled by bit 5 of the Peripheral Control Register. CB2 then goes high if this bit is set to 1, and low if it is reset to 0. The procedure is illustrated in the program below, which repeatedly switches CB2 between its two states, with a short delay.

Example 13.8

```
100 REM      ****************
110 REM      *CB2OUTPUT BLINK*
120 REM      *EAF 1982       *
130 REM      ****************
140
150 PRINT"□REMEMBER TO OPEN THE HANDSHAKE ISOLATING SWITCHES!"
160
170 POKE59467,PEEK(59467)AND227:REM     DISABLE SHIFT REGISTER FUNCTION
180 POKE59468,PEEK(59468) AND 31 OR 192:REM       SET CB2 LOW
190 FOR I=1 TO 100:NEXT I
200 POKE59468,PEEK(59468) AND 31 OR 224:REM       SET CB2 HIGH
210 FOR I=1 TO 100:NEXT I
220 GOTO180
```

13.12 SHIFT REGISTER OUTPUT

As previously described, if the shift register is enabled, its output is fed directly to CB2, at a rate which can be determined by the count loaded into the timer. The shift register can be used in a number of ways, but most are useful only when used with machine-code programs. So far as the BASIC programmer is concerned, only the free-running mode described above is useful.

The SR is enabled in its free-running mode by setting bits 4, 3 and 2 of the Auxiliary Control Register (ACR) to 1, 0, 0 respectively. When this is done, the bit shifted out of bit location 7 of the SR is fed to CB2 at the same time as it is fed back into bit 0 of the SR. The number in the SR is shifted one bit every time the timer completes its count. CB2 can thus be fed with a repetitive train of rectangular pulses. If the output from CB2 is then fed to a simple amplifier, musical and other sound effects can be obtained. If an audio amplifier is already available, this may be used. If not, the following circuit,

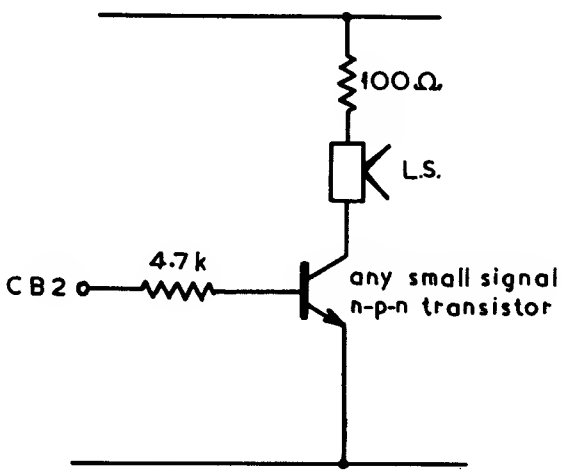


Figure 13.12

which can be powered from the cassette port, is adequate. Almost any small-signal n-p-n transistor can be pressed into service; a p-n-p device may also be used if the circuit is suitably re-arranged.

Any desired number can be loaded into the shift register (memory location 59464), and the counter can similarly be loaded with any desired number from 1 to 255, by POKEing memory location 59464.

It should be noted that once this is done, the 6522 will continue to deliver a continuous output to the CB2 line indefinitely, without any further attention from the microprocessor. The PET can therefore carry out other operations at the same time as generating this audio ouput. This may be seen in the familiar 'Space Invaders' game.

The highest possible frequency (which is quite inaudible) is obtained by loading the counter with 1, and the shift register with alternating 0's and 1's. This may be done by POKEing memory location 59466 (the SR) with 85 (01010101) or 170 (10101010). This produces a signal which changes from high to low or vice versa with every clock pulse – i.e. at a rate of 500kHz.

The lowest possible frequency is obtained by loading the counter with 255 and the shift register with 15 (00001111) or 240 (11110000). This is clearly slower by a factor of 255 × 4, or 1020, so that the frequency will be 490.2 Hz.

Between these two values a whole range of frequencies is available, so that it is quite possible to use the PET as a rather crude electronic organ. It should be noted that the frequency of the signal is extremely precise and stable, since it is derived directly from the PET's crystal oscillator. The PET can therefore be used for technical purposes as a pulse or square-wave generator giving great precision of timing.

Some of the possibilities are demonstrated in the programs which follow.

Example 13.9

This allows the user to load any desired numbers (up to 255) into the counter and shift register. The corresponding signal is fed to CB2, and the waveform of the signal is displayed on the screen. This program is particularly instructive, since the effect of changing the contents of the shift register, and hence the output signal waveform, is instantly obvious both to the eye and the ear.

```
100 REM        ****************
110 REM        *CB2 SOUND DEMO*
120 REM        *EAF 1982      *
130 REM        ****************
140
150 PRINT"■THIS PROGRAM GENERATES AN ALTERNATING   OUTPUT SIGNAL ON CB2. ";
160 PRINT"IT USES THE BUILT-IN SHIFT REGISTER AND COUNTER OF THE    6522,";
170 PRINT" ALONG WITH THE CRYSTAL-CONTROLLED SYSTEM CLOCK
180
190 REM        ENABLE SHIFT REGISTER
200 POKE59467,PEEK(59467) AND 227 OR 16:REM    SET ACR BITS 2&3 TO 0,BIT 4 TO 1
210
220 PRINT"■INPUT A NUMBER (0-255) TO SET COUNTER"
230 INPUT C
240 POKE59464,C:REM       SET COUNT IN TIMER 2 OF 6522
250 INPUT"■INPUT A NUMBER TO BE LOADED INTO THE    SHIFT REGISTER";N
260 N$="":NN$=""
270 FOR J=7 TO 0 STEP -1
280 NN$(J)=STR$(0):N$(J)=""
290 IF N AND 2↑J THEN NN$(J)=STR$(1):GOSUB 470
300 IF NOT N AND 2↑J THEN GOSUB 430
310 NN$=NN$+NN$(J):N$=N$+N$(J)
320 NEXT J
330 PRINT"■BINARY EQUIVALENT IS "NN$"■"
340 PRINT"WAVEFORM IS    ";:IF VAL(NN$(7))=1 THEN PRINT " |";
350 PRINTN$;:IF VAL(NN$(0))=1 THEN PRINT"| ";
```

```
360 POKE 59466,N:REM       LOAD SHIFT REGISTER
370 PRINT:PRINT
380 PRINT"PRESS SPACE BAR TO STOP; ANY OTHER KEY  TO INPUT A NEW NOTE"
390 GET A$:IF A$="" THEN GOTO 390
400 IF A$=" " THEN GOTO 420
410 PRINT"□":GOTO 200
420 END
430 IF J=7 THEN N$(J)="_":GOTO 460
440 IF NN$(J+1)=NN$(J) THEN N$(J)="_":GOTO 460
450 N$(J)="└"
460 RETURN
470 IF J=7 THEN N$(J)="¯":GOTO 500
480 IF NN$(J+1)=NN$(J) THEN N$(J)="¯":GOTO 500
490 N$(J)="┌"
500 RETURN
```

Example 13.10

In this program the shift register is loaded with a fixed number, 15, to give a square-wave output, while the timer is loaded with the ASCII value of a character from the keyboard. Every time a different key is pressed, a new note is produced, so that with some care simple tunes can be played on the PET keyboard.

```
100 REM       PET KEYBOARD MUSIC
110 REM       ******************
120
130 PRINT"□PET KEYBOARD MUSIC PROGRAM"
140 PRINT"PRESS KEYS IN TURN TO PRODUCE NOTES"
150 PRINT"PRESS RETURN TO STOP"
160
170 GET A$:IF A$="" THEN GOTO 170
180 IF A$=CHR$(13) THEN GOTO 230
190 POKE 59467,PEEK(59467)AND 227 OR 16:REM   SET SR TO FREE-RUNNING MODE
200 POKE 59466,15:REM   LOAD SR WITH SQUARE WAVE
210 POKE 59464,ASC(A$):REM      SET TIMER
220 GOTO 170
230 POKE 59467,PEEK(59467)AND 239     :REM    DISABLE SR
240 END
```

Example 13.11

The principle is extended in the next program by loading the counter with numbers read in by means of DATA statements. If these are carefully chosen, quite presentable music can be produced.

```
100 REM         PET MUSIC FROM DATA
110 REM         ********************
120
130 PRINT"□PET MUSIC PROGRAM"
140 PRINT"PRESS ANY KEY TO STOP"
150
160 POKE 59467,PEEK(59467)AND 227 OR 16:REM   SET SR TO FREE-RUNNING MODE
170 POKE 59466,15:REM   LOAD SR WITH SQUARE WAVE
180 READ A:IF A=1 THEN RESTORE:GOTO 180
190 POKE 59464,A:FOR J=0 TO 100:NEXT J
200 GET A$:IF A$="" THEN GOTO 180
210 POKE 59467,PEEK(59467)AND 239     :REM    DISABLE SR
220 END
230 DATA 79,70,58,58,0,58,0,0,70,79,70,79,95,79
240 DATA 0,0,79,70,58,58,0,58,0,0,70
250 DATA 79,70,79,95,79,0,0,106,97,106,119,119,119,
260 DATA 0,0,105,96,105,118,118,96,105,118,118,100,96,0,0,0,1
```

The programs presented here form only an introduction, and the user will doubtless be able to devise many variations on the basic theme.

WARNING When the shift register is enabled in the cyclic mode, it will be found that programs cannot be SAVEd. If the SR is enabled in this way during the course of a program, then it should be disabled again before exiting from the program, by clearing bit 4 of the ACR.

If the program is interrupted by means of the RUN/STOP key, then the SR should be disabled by the immediate mode command

```
POKE59467,PEEK(59467)AND239
```

or more simply

```
POKE59467,0
```

before attempting to SAVE the program (or indeed any other). Alternatively, of course, the ACR may be reset by the usual rather drastic process of turning the machine off and on again.

IF THE SHIFT REGISTER IS NOT DISABLED IN THIS WAY BEFORE ATTEMPTING TO SAVE A PROGRAM ON DISK, ANY OTHER PROGRAMS ALREADY ON THE DISK MAY BE RENDERED UNREADABLE.

CHAPTER 14

Loading and Running Machine-Code Programs

14 Loading and Running Machine-Code Programs

INTRODUCTION

It is sometimes found that the PET is just not able to operate quickly enough to perform a particular task. This may occur, for example, when trying to change the screen display rapidly in a graphics program or when trying to use the user port to time fast-moving objects. In the majority of cases this problem is caused by the time it takes the PET to recognise and obey BASIC instructions. Thus, each time a BASIC instruction is encountered, the BASIC 'book of rules' inside the PET must be consulted so that the instruction can be recognised and the correct action taken. The BASIC instruction, when interpreted, becomes a set of coded binary numbers which the 6502 microprocessor at the heart of the PET can understand. It would clearly be much quicker if the program instructions could be written directly in this binary coded form instead of in the high-level language BASIC. This is the basis of machine-code programming. The resulting increase in speed is spectacular. In some cases the PET will execute a machine-code program several hundred times faster than the equivalent program written in BASIC.

It is not intended here to become involved with the intricacies and techniques of machine-code programming because the subject is vast. The reader is referred to one of the many books on 6502 assembly language programming which are available. However, it is helpful to examine how a machine code program may be loaded into a PET and then run. There are various ways in which this may be achieved.

14.1 DIRECT POKEING FROM BASIC

The BASIC commands POKE and PEEK are very powerful in that they can access *any* of the Random Access Memory (RAM) locations used by the PET, whereas the other instructions operate only in that section of memory reserved for BASIC programs. Thus, the POKE command can be used to POKE a machine-code instruction into any suitable location. It is obvious that this technique should be used with great care. Large sections of the memory are used to control the internal workings of the PET and, should a rogue instruction be POKEd into such a location, it is likely that the machine would be totally disabled. In such cases, the only solution is to switch off and then on again for a fresh start losing, of course, any existing program in the machine in the process. However, there are some regions of the memory which are relatively safe and secure. The most commonly used of these is the space reserved for the operation of the second cassette. Since this is only rarely used these locations are commonly regarded as 'fair game'. The second cassette buffer is placed between locations 826 and 1017 in decimal i.e. between $033A and $03F9 in hexadecimal.

The use of hexadecimal notation is an important technique in machine-code programming. As the name implies, hexadecimal is a number system with a base of 16, the decimal numbers 10, 11, 12, 13, 14, 15 being represented by the alphabetic characters A, B, C, D, E, F respectively. The use of hexadecimal greatly simplifies the tedious problem of writing machine-code instructions in binary notation. For example, suppose we wish to write the binary instruction 11101010. Such a group of eight binary digits is called a BYTE. In hexadecimal this byte would be written simply as EA (i.e. $E \times 16 + A \times 1 = 14 \times 16 + 10 \times 1 = 234$). The hexadecimal version is clearly much less laborious to use and is less prone to error. The PET simply takes the hexadecimal characters and converts them into the binary equivalent for its own use. (Thus E = 1110 and A = 1010 so EA = 11101010.)

Conversely, on output, the PET converts from binary to hexadecimal for the convenience, and sanity, of the user. In order to be able to identify hexadecimal numbers when printed or written they are usually preceded by the $ sign.

The POKE address and argument must always be in decimal. This can be a nuisance if the user is more familiar with hexadecimal notation because the corresponding decimal numbers will be unfamiliar. (This can be overcome by the use of more sophisticated loader programs, as discussed later.)

Having placed the program in some safe region of the memory, the PET must now be directed to the start location in order to run the program. It is not possible to use the RUN command because this only operates on a BASIC program. The most convenient command to use is SYS. Thus, if the start location of the machine code program is in decimal location 826, then the BASIC command SYS 826 will start the program. The program will continue to run until a special instruction in the program returns the PET to BASIC operation. This command is the 'Return from Subroutine' command (code 96 in decimal or $60 in hexadecimal). If the original SYS command was a direct (i.e. keyboard) command then, on return to BASIC, the normal READY response will be displayed. If the SYS was part of a BASIC program then, on returning to BASIC, the next instruction of the BASIC program will be obeyed.

Consider the following simple program:–

Example 14.1

```
10 POKE 826,234
20 POKE 827,96
30 SYS 826
40 END
```

In line 10 the first location in the second cassette buffer is loaded with 234, i.e. $EA. Line 20 loads the next location with 96, or $60. Line 30 is the SYS command pointing the PET to the first of the machine-code instructions. On RUNning the program the machine-code instructions are loaded and the SYS command causes the instruction in 826 to be obeyed. This instruction is, in fact, a No Operation instruction which has no effect at all (just as a BASIC REM has no effect on the running of the program). The next instruction, in 827, is a Return from Subroutine command and returns control to BASIC where the next instruction is END. Thus the sum total effect of this program is to do nothing! However, it does show how machine-code programs can be handled. Note that the machine-code needs to be loaded only once. The locations holding the program are not affected by the BASIC command NEW nor, if chosen correctly, by the loading of another BASIC program from disk or cassette. Thus, after being loaded, the machine-code instructions may be called repeatedly within the same BASIC program or even from other BASIC programs. Only re-POKEing the locations or switching off the PET will disturb them.

If the second cassette buffer is required during a program it is then, of course, not possible to use these locations to house any machine-code instructions. A useful technique in such cases is to create a reserved region at the top of PET's memory. This is done by fooling the PET's operating system into thinking that the available memory is smaller than it really is. The address of the highest available memory location is stored in the PET in locations 52 ($34) and 53 ($35). The higher order byte of this address is stored in location 53 and the lower order byte in location 52. Therefore, if we change either of these numbers, the apparent size of the available memory will be altered. By this means we can create our own protected area at the top of memory which will not be used by the PET's BASIC interpreter.

For instance, a protected area of 256 bytes will be created if the higher order byte of this upper RAM pointer is reduced by 1. It is not necessary to know the old value of this pointer. It may be

reduced by the required amount simply by using a combination of PEEK and POKE commands:

```
POKE 53,PEEK(53)-1
```

The lowest reserved address is then readily calculable in decimal form: it will be:

```
PEEK(53)*256+PEEK(52)+1
```

It can be seen that any desired amount of RAM space (within the limits set by the PET's memory size) may be reserved by this method provided that a reasonable amount of RAM is left for use by the BASIC interpreter.

14.2 THE USE OF LOADER PROGRAMS

A loader program is simply a development of the method of using POKE to place machine-code instructions in the memory. This type of program may either be called a machine-code loader, because it loads machine-code, or it may confusingly be called a BASIC loader because it loads *from* a BASIC program. Instead of POKEing each instruction individually the loader program handles all the machine-code instructions as a block of DATA statements. Each machine-code instruction is READ and POKEd sequentially into the memory, starting at a specified location.

Example 14.2

```
10 GOSUB 1000
20 SYS 832
30 END
1000 READ S:REM S=START LOCATION
1010 READ A:IF A<0 THEN RETURN
1020 IF A>255 THEN 1040
1030 POKE S,A:S=S+1:GOTO 1010
1040 PRINT "BYTE"S"=["A"] ???":END
1050 DATA 832
1060 DATA 120,169,79,133,144,169,3,133,145,169,1
1070 DATA 133,2,88,96,165,151,197,0,240,9
1080 DATA 133,0,169,16,133,1,76,65,228,201,255
1090 DATA 240,249,165,1,240,4,198,1,208,241
1100 DATA 198,2,208,237,169,4,133,2,169,0,133,151
1110 DATA 169,2,133,168,208,223,-1
```

This program may be used to provide a 'repeat' facility when any of the PET's keys is held down. It is a good example of the type of machine-code program which can be loaded and forgotten while the user continues to operate the machine in BASIC, load in other BASIC programs etc, while still employing the machine-code program. The operation of the loader part of the program is as follows. Line 10 calls the loader sub-routine which starts at line 1000. Here the value of S is READ from the data and this gives the starting location for the machine code program. In 1010 the first instruction byte is READ into variable A. If this byte is found (line 1020) to be valid it is POKEd (line 1030) into the location specified by S; if it is invalid the program ends with an error message (1040). The value of S is then incremented by 1 and the process is repeated for the next byte. If the exact number of data bytes in the program is known, the loading operation can be controlled by a FOR . . . NEXT loop. A more flexible technique is to make the process open-ended and terminate the operation with a terminating character at the end of the data. This is achieved here by the introduction of a negative number in the last data position, causing a RETURN to be executed from line 1010. The machine

code instructions are now in place and it only remains to call the program with a SYS (in line 20) to the starting location. (N.B. This program was written for a 4000-series machine. To make it run on a 2000 or 3000 series machine it is necessary to replace the two circled bytes with the decimal numbers 46 and 230 respectively.)

The main disadvantage of the above program is that the instructions and locations must be in decimal. This can be a serious disadvantage because one becomes familiar with the hexadecimal version of a program as it is being developed and modified, but once all the bytes have been converted into decimal it becomes unrecognisable. A solution is to use a version of the loader which automatically converts from hexadecimal to decimal before the bytes are POKEd.

Example 14.3

```
10 GOSUB 1000
20 SYS 832
30 END
1000 READ S
1010 READ A$:L=LEN(A$):IF A$="*"THEN RETURN
1020 IFL<1 OR L>2 THEN 1090
1030 N=0:IF A$>"" AND LEN(A$)<3 THEN1050
1040 GOTO1090
1050 FOR J=1 TO LEN(A$):X=ASC(MID$(A$,J,1))-48
1060 N=N*16+X+(X>9)*7:NEXT J
1070 IF N<0 OR N>255 THEN1090
1080 POKE S,N:S=S+1:GOTO1010
1090 PRINT"BYTE"S"=["A$"] ???":END
1100 DATA 832
1110 DATA 78,A9,4F,85,90,A9,03,85,91,A9,01
1120 DATA 85,02,58,60,A5,97,C5,00,F0,09
1130 DATA 85,00,A9,10,85,01,4C,55,E4,C9,FF
1140 DATA F0,F9,A5,01,F0,04,C6,01,D0,F1
1150 DATA C6,02,D0,ED,A9,04,85,02,A9,00,85,97
1160 DATA A9,02,85,A8,D0,DF,*
```

It can be seen that the data statements in lines 1110 to 1160 are now written as hexadecimal bytes and the program is quite recognisable to anyone who is familiar with it in hexadecimal form so that, as a result, the program may be modified easily by modifying the data. The operation of the loader is very similar to that of the previous example. The data are now treated as strings and the terminator is a string character, a "*". The main interest lies in lines 1050 and 1060 where the hexadecimal bytes are converted into their decimal equivalents. In 1050 a FOR . . . NEXT loop is set up of length equal to the number of characters in the hexadecimal byte. (This is always equal to 2 if the data is written as in this example but, in fact, the leading zero in, for example, 01 or 04, could be omitted.) The ASCII value of the first character of the byte is then determined, 48 is subtracted from it and this number is assigned to the variable X. This is because the ASCII values for the numerals 0 to 9 start at the value 48 so that subtracting 48 from the code gives the numerical value. In 1060, on the first cycle of the FOR . . NEXT loop, N is set to this value. However, a problem occurs if the character is one of the special hexadecimal characters A to F. The ASCII value for A does not follow on immediately from the ASCII value for 9 but instead there is a gap of 7 between them. Thus ASCII 9 = 57 but ASCII A = 65. This means that in these cases the value of X will be too large by 7. The solution is to employ a logical operator. Thus, in line 1060, X is added to N and the term (X>9)*7 is also added. The term in brackets has the property of being zero if X is not greater than 9 and being –1 if X exceeds 9. Thus, in the case where the character is a numeral between 0 and 9 this term is zero, but when one of the letters

A to F is encountered a further 7 is subtracted from the ASCII value. At the end of line 1060 the FOR ... NEXT loop is re-entered to convert the second character of the byte (if present). When line 1060 is reached again, the old value of N is again multiplied by 16 after which it is added to the computed value of the second character. After the final value of N has passed its validity test in line 1070, it is POKEd into its location and the next byte is READ in.

This loader is a useful device for loading machine code from BASIC and the technique is very widely used. Its main disadvantage is that it is rather slow compared with the decimal loader and this can be annoying when long machine-code programs have to be loaded, because the PET is effectively disabled during this period. (N.B. This program was written for a 4000 series machine. To make it run on a 2000 or 3000 series machine it is necessary to replace the two circled bytes with the values 2E and E6 respectively.)

14.3 THE TERMINAL INTERFACE MONITOR (TIM)

The method of POKEing instructions into the PET's memory is effective, especially when performed by a loader program, but it has the disadvantage that it is difficult to make any changes without re-loading the entire program. The only way that a location can be checked from BASIC is to use the PEEK command, and only one location at a time can be examined with a single PEEK.

The firmware of the PET includes an extensive program which, among other things, allows the contents of any memory location to be inspected and, if required, to be modified. The program is called the Terminal Interface Monitor known, somewhat sinisterly, as The Monitor or more familiarly as TIM.

You will probably have met the monitor already. It has a habit of appearing, uncalled-for, in the middle of BASIC runs as a result of some error on your part. In this context the appearance of the monitor display usually heralds disaster. However, when under control, the monitor can be very useful.

To call up the monitor intentionally, type SYS 4 and press RETURN. The monitor display will appear and will look something like:–

```
B*
     PC   IRQ   SR AC XR YR SP
.;  0005 034F 30 00 5E 04 F6
.
```

DON'T PANIC! This display is quite normal. The monitor is simply showing you the contents of the registers in the 6502 microprocessor at the heart of the PET. Thus, PC is the Program Counter, SR the Status Register, AC the accumulator, etc. Note that, when the monitor is in action, *all* numbers must be in hexadecimal form.

Once you have invoked the monitor, the prompt signal is the full stop. This can be seen at the bottom-left of the above display. It is usually followed by the cursor, inviting an input from the keyboard. A number of single-letter commands are available to you when in the monitor:–

M	display content of locations specified
R	display registers
G	execute machine code program from specified address
X	exit to BASIC
L	load program (from cassette or disk)
S	save program (on to cassette or disk)

Probably the most useful command is M, the 'display memory' command. Thus, if you type:–

M space start address, end address

followed by RETURN

e.g. .M 033A,034F

the monitor will respond with a display which will look something like:–

```
.M 033A,034F
.:  033A 01 00 00 00 01 00 00 FF
.:  0342 2A 00 00 00 00 00 00 00
.:  034A 00 00 00 00 00 00 00 00
.
```

The characters on the left of the display, e.g. 033A, 0342, 034A are the locations being examined. The starting location and every 8th location thereafter are labelled but, in fact, the contents of every location in the specified range are displayed. Thus, on the first line, the contents of 033A to 0341 are, respectively, 01, 00, 00, 00, 01, 00, 00, FF. In fact, the contents of the specified locations don't look very exciting. This is because no specific program has been placed in this part of memory since switch-on. If we look again at the same locations after a program has been loaded they may look quite different:–

```
.M 033A,034F
.:  033A D0 F4 EE A1 03 D0 EF EE
.:  0342 A2 03 4C 30 03 EC 41 E8
.:  034A F0 10 EE B0 03 D0 DF EE
.
```

Once the contents of a memory location or a register are on display they can be changed very easily. The cursor is moved, using the usual cursor control keys, until it is over the offending data and the new data are overtyped. When the modifications to a given line are complete the RETURN key is pressed, which writes in the new data.

It is not necessary to return to BASIC to execute a program; it can be run directly from the monitor by typing:–

G space start address

followed by RETURN

e.g. .G 033A

Programs which are to be executed like this must terminate with a BREAK instruction ($00) rather than a Return from Subroutine ($60), to leave the user in the monitor at the end of the program.

A program may be SAVEd directly from monitor using the command:–

.S space "progname", 08, start address, end address PLUS 1

followed by RETURN

e.g. .S "TEST", 08, 033A, 0350

will save on disk the program located between $033A and $034F. This will be saved on disk as normal under the program name TEST but it will be a machine code program. To SAVE on cassette the 08 in the above example *must* be replaced with 01.

The machine code program so SAVEd may be LOADed from monitor with the single instruction:–

```
.L "TEST",08
```

(Again, 01 should be substituted for 08 for cassette operation. Alternatively, when LOADING, omission of the device number gives cassette operation by default.)

It is also possible to LOAD the machine-code program without using the monitor by making use of the normal disk or cassette LOAD commands. The program will automatically be located in the correct position in memory.

14.4 THE USE OF AN ASSEMBLER

An assembler is a program which can be resident in firmware or on disk and allows the use of a type of programming language, which is of a slightly higher level than machine-code, called a mnemonic assembly language. Using such a language means that every possible machine-code instruction can be represented by a unique three-letter mnemonic. For example, $EA, the No Operation instruction referred to earlier, is represented by the mnemonic NOP; $60, a Return from Subroutine, is represented by RTS; $00, Break is represented by BRK etc. The complete program may be written in this form and then fed to the assembler which converts the mnemonics into the corresponding machine-code instructions. The advantage of this is that it is much easier to construct programs using the helpfully-spelt mnemonics than it is using the completely arbitrary machine codes, whether in hexadecimal, binary, or decimal. A further important advantage is that the assembler allows you to use labels within the program so that the wearisome task of calculating the addresses for jumps and branches in the program is undertaken by the assembler, not by you.

Finally, having assembled the complete program in machine-code, the assembler loads the program into the memory locations specified so that all you have to do is RUN it, either from BASIC or from the monitor.

This is, without doubt, the easiest and most effective way to develop and load machine-code programs and if lengthy programs are to be written an assembler is an essential programming tool.

14.5 ELEMENTARY MACHINE-CODE EXERCISES

The following elementary machine-code exercises are not presented as tuition in machine-code programming but simply as vehicles to demonstrate the LOADing and RUNning techniques discussed in this chapter. In the exercises a few of the commands of the 6502 instruction set are employed to perform some simple arithmetic operations. The programs may be POKEd manually from the keyboard, they may be POKEd from a BASIC program, they may be written and run from the monitor, or assembled and loaded with an assembler. Note that, if the monitor command .G is used to run the programs, the RTS instruction at the end of each program must be replaced by a BRK instruction, $00.

WARNING When a machine-code program is run, any error in the program is likely to lock-up the PET completely. Control can usually be recovered only by switching the machine off and on again. This of course destroys any program in memory so, before running any machine-code program, it is wise to SAVE the program or at least to write it down!

The following programs may be inserted at any convenient point within the memory space of the PET. Probably the simplest method is to make use of the second cassette buffer, which occupies 192 memory locations extending from $033A-$03F9 (826 to 1017 decimal). For simplicity the addresses used below start from $033A, but the start address may of course be changed at will.

Exercise 14.1

This program reads a number (here $AB) directly into the accumulator and then writes it into memory, at the arbitrarily-selected location $03F9 (1017 decimal).

Address	*Machine-Code Instruction*	*Mnemonic*
$033A	A9 AB	LDA #$AB
$033C	8D F9 03	STA [$03F9]
$033F	60	RTS

The program may be stored at the locations shown and then run using the appropriate SYS command. Remember that all the numbers are in hexadecimal. If the monitor is used to load the program, no difficulty arises, but if the program is to be loaded using POKE commands, both the addresses and the instructions must be converted to decimal form. Note that addresses in machine-code are written low-order byte first. Thus 03 F9 becomes F9 03.

Exercise 14.2

In this exercise the number is read into the accumulator from a specified memory location (here $03F0) and written into location $03EA.

$033A	AD F0 03	LDA [$03F0]
$033D	8D EA 03	STA [$03EA]
$0340	60	RTS

Check the operation of the program by POKEing different numbers (up to 255) into location $03F0 (1008 decimal) and PEEKing location $03EA (1002 decimal), after the program has been run, to confirm that they have been correctly transferred.

Exercise 14.3

Here a number is read from memory location $03F6 (1014 decimal) and loaded into the accumulator. A second number, read from memory location $03F7 (1015), is added to it and the result is stored in memory location $03F8 (1016). The carry flag must first be cleared, since if it is set to 1 the microprocessor will assume that there has been a carry from a previous addition and add this into the sum. The operation is:

$$A + M + C \rightarrow A, C$$

$033A	AD F6 03	LDA [$03F6]
$033D	18	CLC
$033E	6D F7 03	ADC [$03F7]
$0341	8D F8 03	STA [$03F8]
$0344	60	RTS

Note that none of the numbers, including the sum, must exceed a value of $FF (255).

Exercise 14.4

We may similarly subtract two numbers. In this case the carry must first be set to 1 since the SBC (subtract with borrow) operation is:

$$A - M - \overline{C} \rightarrow A, \overline{C}$$

Note that, if the subtrahend is greater than the minuend, the (negative) answer will be expressed in complement form and will be difficult to interpret.

```
$033A    AD E0 03    LDA  [$03E0]
$033D    38          SEC
$033E    ED E1 03    SBC  [$03E1]
$0341    8D E2 03    STA  [$03E2]
$0344    60          RTS
```

Here, the contents of memory location $03E1 (993) are subtracted from those of location $03E0 (992) and the result is stored at $03E2 (994).

Exercise 14.5

In this program the mean value of two numbers is calculated and rounded down.

```
$033A    18          CLC
$033B    AD E6 03    LDA  [$03E6]
$033E    6D E7 03    ADC  [$03E7]
$0341    6A          ROR
$0342    8D E8 03    STA  [$03E8]
$0345    60          RTS
```

The numbers to be averaged are read from locations $03E6 (998) and $03E7 (999) and their mean is stored at $03E8 (1000).

The SED and CLD Instructions

The PET's microprocessor, the 6502, may be set to add or subtract decimal rather than hexadecimal numbers, by means of the instruction SED (F8). The processor is returned to the hexadecimal mode by the instruction CLD (D8).

N.B. It is in general good practice to set the processor specifically to the desired mode within every program you write, since it is always possible that some previous program has left it set to the wrong mode.

Exercise 14.6

Modify the programs of exercises 14.3 and 14.4 for the addition of decimal, rather than hexadecimal, numbers. Verify that the modified programs work as predicted.

The NOP Instruction

Where a machine-code program is subject to modification as in Exercise 14.6, the NOP or 'no operation' instruction (EA) will often be found useful. It may be used to fill unused memory locations within a program since the processor does nothing on receipt of a NOP instruction, simply advancing to the next instruction in sequence.

Exercise 14.7: Branches and loops

This program converts a two-digit decimal number, stored in location $03E0 (992), to a hexadecimal number stored in location $03F0 (1008).

The decimal number is first copied into location $03E2 (994) and then progressively decremented. Simultaneously the number stored in location $03F0 (initially 0) is correspondingly incremented. At each step the value of the number remaining in $03E2 is checked. So long as this value is greater than zero, the program loops back and repeats the process. Once its value has been reduced to zero, the program exits from the loop by means of a branch instruction.

Location		*Instruction*	*Mnemonic*	*Comment*
$033A		AD E0 03	LDA [$03E0]	
$033D		8D E2 03	STA [$03E2]	Copy decimal number
$0340		A9 00	LDA #00	Clear accumulator
$0342		8D F0 03	STA [$03F0]	Clear hex count
$0345	LOOP	D8	CLD	Clear decimal mode
$0346		EE F0 03	INC [$03F0]	Increment hex count
$0349		F8	SED	Set decimal mode
$034A		AD E2 03	LDA [$03E2]	Get decimal number
S034D		38	SEC	Set carry flag
$034E		E9 01	SBC #01	Subtract 1
$0350		8D E2 03	STA [$03E2]	Save decremented decimal number
$0353		D0 F0	BNE LOOP	Return to start of loop if decimal number ≠ 0
$0355		D8	CLD	Clear decimal mode
$0356		60	RTS	

APPENDIX A

BASIC COMMANDS AND STATEMENTS

COMMAND/ STATEMENT	EXAMPLE	PURPOSE
ASC	10 A = ASC("XYZ")	Returns integer value corresponding to ASCII code of first character in string.
CHR$	10 A$ = CHR$ (N)	Returns character corresponding to ASCII code number.
CLR	CLR	Sets variables to zero or null.
CMD	CMD 5	Keeps open IEEE device specified in logical file, to monitor bus.
CONT	CONT	Continues program execution after a STOP command.No program changes allowed.
CLOSE	10 CLOSE L	Closes logical file L.
DATA	10 DATA 1,2,3,4	Specifies data to be read from left to right.
	20 DATA TOM,SUE	Purely alphabetic strings do not need to be enclosed in quotes.
	30 DATA TOM," ", CAT	If string contains spaces, commas, colons, or graphic characters, the string must be enclosed in quotes.
DIM	10 DIM A(n)	Specifies maximum number of elements in an array or matrix.
	20 DIM A(n,m,o.p)	Specifies maximum number of elements in each dimension of a multi-dimensional array.
	30 DIM A(n), B(m)	Format for dimensioning two arrays.
	40 DIM A(N)	Format for dynamic dimensioning.
	50 DIM A$(n)	Format for string arrays.
END	999 END	Terminates program execution.
FRE	PRINT FRE(0)	Returns number of bytes of available memory.
FOR . . . NEXT	10 FOR A = 1 TO 20	Loop control. Performs all functions between FOR and NEXT as many times as specified by index.
	90 NEXT A	In this example, the index variable is A.
GET	10 GET C	Accepts single numeric character from keyboard.
	20 GET C$	Accepts single string character from keyboard.
	30 GET#L,C	Accepts single character from specified logical file.
	40 GET#L, C$	Accepts specified single string character from logical file.
GOSUB	10 GOSUB n	Begins execution of a subroutine which begins on a specified line.
GOTO	GOTO n	Continue program execution at line n after a STOP command. Program changes are permitted.
	10 GOTO n	Transfers control (jumps) to specified line, skipping over intervening lines.
INPUT	10 INPUT A	Accepts value of A from keyboard.
	20 INPUT A$	Accepts value of string variable A from keyboard The string does not have to be enclosed in quotes.
	30 INPUT A,A$,B.B$	Accepts specified variables from keyboard.
	40 INPUT#L,A	Accepts value of A from logical file L.
	50 INPUT#L,A$	Accepts specified string from logical file L.
	60 INPUT#L,A,A$,B,B$	Accepts specified variables from logical file L. Strings do not have to be enclosed in quotes.
IF . . . THEN	10 IF A=10 THEN PRINT A	If condition is 'TRUE', instruction following 'THEN' (in this example, 'PRINT A') is executed. Otherwise, the next statement in sequence is executed.
IF . . . GOTO	10 IF A=1 GOTO n	If condition is true, control is transferred to specified line. Otherwise, the next statement, following the IF . . . GOTO, is executed.
LEFT$	10 ?LEFT$(X$, A)	Returns leftmost A characters from string.
LEN	10 ?LEN(X$)	Returns length of string.
LET	LET A = 2	Assign a value to a variable.

COMMAND/ STATEMENT	EXAMPLE	PURPOSE
LIST	LIST	Lists current program.
	LIST –n	List current program up to line n.
	LIST n-m	Lists lines n to m of current program.
	LIST n-	Lists current program from line n to end.
LOAD	LOAD	Loads next encountered program from tape unit.
	LOAD "NAME"	Loads program NAME from tape unit.
	LOAD "NAME", D	Loads program NAME from device D.
MID$	10 ?MID$(X$,A,B)	Returns B characters from string, starting with the A th character.
NEW	NEW	Deletes current program from memory, sets variables to zero.
ON . . . GOTO	10 ON A GOTO m,n,r	Transfers control to specified line (in this example m, n or r, depending on value of index A).
ON . . . GOSUB	10 ON A GOSUB m,n,r	Begins execution of subroutine which begins on line m, n or r, depending on the value of index A.
OPEN	10 OPEN L	Opens logical file L for read only from tape unit.
	20 OPEN L,D	Opens logical file L for device D.
	30 OPEN L,D,C	Opens logical file L for command C from device D.
	40 OPEN L,D,C, "NAME"	Opens logical file L on device D; if device D accepts formatted files, file NAME is positioned for command.
PEEK	PEEK(A)	Returns byte value from address A.
POKE	POKE A,B	Loads number B into address A.
POS	10 PRINT POS(0)	Prints next available print position of cursor on screen. (Dummy argument – may be string or numeric.)
PRINT	10 PRINT A	Prints value of A on display screen.
	20 PRINT A$	Prints specified string on screen.
	30 PRINT A,A$	Prints specified values of variables on screen, beginning in next available print position (pre-TABbed positions are in columns 0, 10, 20, 30 etc).
	40 PRINT A;A$	Prints specified values and strings on screen separated by 3 spaces if numeric, concatenated if strings.
	50 PRINT#L,A	Prints specified value to logical file L.
	60 PRINT#L,A$	Prints specified string to logical file L.
READ	10 READ A	Obtains value of A from DATA statement.
	20 READ A$	Obtains string value for A$ from DATA statement.
	30 READ A,A$,B,B$	Obtains specified values for strings and numeric variables from DATA statements.
REM	10 REM **COMMENT**	Inserts non-executable comments in a program for documentation purposes.
RESTORE	10 RESTORE	Permits re-reading of DATA statements without re-running program.
RETURN	1000 RETURN	Subroutine exit; transfers control to the statement following most recent GOSUB directing transfer to the subroutine.
RIGHT$	10 ?RIGHT$(X$,A)	Returns rightmost A characters from string.
RUN	RUN	Begins execution of program at lowest line number.
	RUN n	Begins execution of program at line n. (Resets variables.)
SAVE	SAVE	Saves current program on tape unit.
	SAVE "NAME"	Saves current program NAME on tape unit.
	SAVE "NAME",D	Saves current program NAME on device D.
	SAVE "NAME",D,C	Saves program NAME on device D. C specifies End of File, or End of Tape.
STOP	STOP	Stops program execution (see CONT).

COMMAND/ STATEMENT	EXAMPLE	PURPOSE
SYS	SYS(X) or SYSX	Complete control of PET is transferred to a machine code routine at decimal address contained in the argument.
SPC	10 SPC(N)	Prints N spaces or blanks.
STEP	10 FOR A= TO 20 STEP 2	Step specifies size of increment to be added to index to increase or decrease its value towards
	90 NEXT A	the specified limit.
STR$	10 A$ = STR$(A)	Returns string representation of number.
TAB	10 PRINT TAB(N);A	Prints value of A in character position N+1 on screen.
	20 PRINT TAB(N);A$	Prints string beginning in character position N+ 1 on screen.
TI$	TI$ = "HHMMSS"	Set PET's internal clock to real time.
TI	PRINT TI	Displays number of "jiffies" since PET was powered up or clock was zeroed. (A jiffy = 1/60 of a second).
USR	USR(X)	Transfers program control to a machine code program whose start address is stored at locations 1 and 2. X is a parameter passed to and from the machine language monitor.
WAIT	WAIT A,B,C	Stops execution of BASIC until contents of location A, ANDed with B and exclusive ORed with C, is not equal to zero. C is optional and defaults to zero.
VAL	10 A=VAL(A$)	Returns numeric representation of string; if string not numeric, returns "0".
VERIFY	10 VERIFY	Verifies most recent program saved on cassette by reading it and comparing it with program still in PET's memory.
	20 VERIFY "NAME"	Verifies specified program NAME saved on cassette by reading it and comparing it with program still in PET's memory.
	30 VERIFY "NAME",D	Verifies specified program NAME saved on device D by reading it and comparing it with program still in PET's memory.

BASIC 4 incorporates a number of additional disk and file handling commands which are discussed in the relevant chapters.

APPENDIX B

ARITHMETIC FUNCTIONS

FUNCTION	EXAMPLE	PURPOSE
ABS	10 C = ABS(A)	Returns magnitude of argument without regard to sign.
ATN	10 C = ATN(A)	Returns arctangent of argument; C will be expressed in radians.
COS	10 C = COS(A)	Returns cosine of argument. A must be expressed in radians.
DEF FN	10 DEF FNA(B) = C*B↑2	Allows user to define a function. Argument B is a dummy variable.
EXP	10C = EXP(A)	Returns constant 'e' raised to power of the argument. In this example, e^A.
INT	10 C = INT(A)	Returns largest integer less than or equal to argument. For example INT(-3.6) returns value of -4.
LOG	10 C = LOG(A)	Returns natural logarithm of argument. Argument must be greater than or equal to zero.
RND	10 C = RND(A)	Generates a random number between zero and one. If A is less than 0 the same random number sequence is produced for a given A. If A equals 0 the sequence of random numbers is generated from a free-running clock and is not repeatable. IF A is greater than 0 then, from power up, the same sequence is produced. (N.B. BASIC 2 operates in a slightly different way).
SGN	10 C = SGN(A)	Returns –1 if argument is negative, returns 0 if argument is zero, and returns +1 if argument is positive.
SIN	10 C = SIN(A)	Returns sine of argument. A must be expressed in radians.
SQR	10 C = SQR(A)	Returns square root of argument.
TAN	10 C = TAN(A)	Returns tangent of argument. A must be expressed in radians.

ARITHMETIC, RELATIONAL AND BOOLEAN OPERATORS

	SYMBOL		EXAMPLE	PURPOSE	
	=		10 A= B	Assigns a value to a variable	
			10 LET A= B	LET is optional	
	/		10 C=A/B	Division	
	*		10 C=A*B	Multiplication	
	+		10 C=A+B	Addition	
	-		10 C=A-B	Subtraction	
	↑		10 B= C↑2.3	Raises variable to specified power.	
Relational Operators	=		A=B	True if A is equal to B	
		e.g.	10 IF A=B THEN GOTO n		
	<>		A <> B	True if A is not equal to B	
		e.g.	10 IF A<> B THEN GOSUB n		
	<		A<B	True if A is less than B	
		e.g.	10 IF A< B THEN C=23		
	>		A>B	True if A is greater than B	
		e.g.	10 IF A>B THEN G$="YES"		
	< =		A< =B	True if A is less than or equal to B	
		e.g.	10 IF A< =B THEN n		
	> =		A> =B	True if A is greater than or equal to B	
		e.g.	10 IF A> =B GOSUB n		
Boolean Operators	AND		A AND B	True if A and B are both true	A and B here must be Boolean variables, with values of either 0 or 1.
	OR		A OR B	True if either A or B is true	
	NOT		NOT A	True if A is not true	

N.B. The Boolean operators may also be applied to numbers or numerical variables; the process is complex and can properly be understood only by a detailed inspection of the binary representations of the numbers concerned. This is demonstrated in Chapter 13.

APPENDIX C

EXTENSION ROM's FOR THE PET/CBM

A number of plug-in ROM's can be purchased, which extend the facilities of the PET in various ways. The following brief notes on a few of these chips may be of interest.

1) *The Programmer's Toolkit*

This chip offers a number of useful additional commands. As well as allowing a program to be traced through either at reduced speed or in single steps, the Toolkit allows data files or program segments read from cassette to be appended to a program already in the machine. Another command causes every line containing a specified string to be listed. Complete sections of program can be deleted, and the lines of a program can be automatically numbered or renumbered. The HELP command, following a syntax error, causes the offending line to be reprinted with the error high-lighted.

The HELP, AUTO (number) and RENUMBER commands alone make the Toolkit well worth the modest purchase price.

2) *The Petmaster Superchip*

This chip can be used on its own, but since the facilities offered are essentially complementary to those of the Toolkit, is best used in conjunction with the latter. A number of useful facilities are offered, including a range of screen-handling facilities very similar to those offered in the 8000-series machines. The character sets may be interchanged by a key stroke, without the tedious POKE 59468, 12 or 14 command. A RETRACE (as opposed to TRACE) facility is offered, allowing the user to back-track from a program error. The PET can also be stopped completely in its tracks; when this is done even the system clock stops counting.

SHRINK removes all REM's and unnecessary spaces in a program, to economise on memory. A number of other facilities are available, including automatic key repeat (invaluable for editing) and single-key entry of 26 BASIC commands. This facility is particularly useful where the machine is used for class demonstrations, but is gratifying to have at all times. These two facilities alone are worth the (rather high) price.

3) *The Pic-Chip*

The Pic-Chip is designed solely for use in conjunction with graphics displays. It does not introduce any new graphics symbols, but allows the existing PET graphics symbols to be simply manipulated in a variety of ways. Since the commands used are rather specialised, being essentially an extension of PET BASIC, the properties of the chip are best seen by running the demonstration programs. The Pic-Chip is a relatively specialised device.

4) *Other add-on ROM's; shortage of ROM sockets*

A number of software packages such as Visicalc now incorporate a plug-in chip (a "dongle") as security against copying. While not strictly an extension ROM in the sense of those described above, such a chip competes with other add-on ROM's for the limited number of sockets available. Various means may be used to overcome this. The ROM's may be "piggy-backed" and controlled by an external switch or switches, or an extension board may be used. A very convenient commercially-produced extension board is the "ROM Pager" produced by Reprodesign Microcomputer Services. This allows up to eight ROM's to be used, the required chip being selected by a software command.

5) *Use of Extension ROM's with the Computhink Disk Drive*

The Computhink disk drive system also competes for some of the ROM space available. Your PET dealer should therefore be consulted before the purchase of any extension ROM, if this is to be used in conjunction with the Computhink drive.

INDEX

INDEX

PROGRAM DISK

A 5¼ inch floppy disk containing all of the main programs listed in this book can be obtained from:

A. R. B. Associates
445 Wilbraham Road,
Chorlton-cum-Hardy
Manchester M21 1US
U.K.

Price £19.95 inc. postage within the United Kingdom and Europe
£22.50 outside the U.K.
(State if 8000-series disk format required) Prices correct at September 1982.